DEPARTMENT OF ENGINEERING

N^{th} root computing methods

D. MARTIN

REPORT NO. 63-18

UNIVERSITY OF CALIFORNIA, LOS ANGELES

Report No. 63-18
April 1963

N^{th} ROOT COMPUTING METHODS

David F. Martin

DEPARTMENT OF ENGINEERING
UNIVERSITY OF CALIFORNIA
LOS ANGELES 24, CALIFORNIA

FOREWORD

The research described in this report, *N^{th} Root Computing Methods*, by David F. Martin was carried out under the technical direction of M. Aoki, B. Bussell, G. Estrin and C. T. Leondes and is part of the continuing program in Digital Technology Research. This report is based on a dissertation submitted in partial satisfaction of the requirements for the degree Master of Science in Engineering at the University of California, Los Angeles.

This project is conducted under the sponsorship of the Department of the Navy, Office of Naval Research and the Atomic Energy Commission. Submitted in partial fulfillment of Contract Number Nonr 233(52).

ABSTRACT

Five main classes of n^{th} rooting methods are discussed in this report. An n^{th} rooting method derivable from the binomial series expansion is developed, and both restoring and nonrestoring versions are treated. For the special case of the binary square root, a nonrestoring version of this method using normalized remainders is simulated and a statistical timing distribution obtained.

Other n^{th} rooting methods discussed are a truncated series method, Euler iteration formulae, extensions of a square root method given by M. Nadler, Padé approximations and the log-exponential method. A particular mechanization of the log and exponential functions developed by Cantor, Estrin, and Turn is compared timewise with the other n^{th} rooting methods. Hardware and storage requirements are considered in all cases.

It is concluded that the log-exponential mechanization of Cantor, Estrin, and Turn is the fastest and most versatile except for very small values of n. The binomial series method is found to be fastest for the binary square root.

ACKNOWLEDGEMENT

This work was supported, in part, by Contract Number Nonr-233(52). The author wishes to thank Professors M. Aoki, G. W. Brown, and G. Estrin for serving as his thesis committee. Special thanks go to Prof. Estrin for many helpful suggestions and generous discussions.

The author also wishes to thank Numerical Analysis Research and the Western Data Processing Center, Dr. G. W. Brown, Director, for use of the IBM 7090 Data Processing System.

TABLE OF CONTENTS

CHAPTER I

Introduction

Important elementary functions rarely included in the basic set of operations of most computers are the integral roots of an operand. In particular, the square root plays an important role in the solution of quadratic equations, phasor algebra, asymptotic expansions of Bessel functions, and a host of other applications. Less frequently required are the higher integral roots. This report concentrates its attention on integral n^{th} roots, with particular emphasis on the square root.

Programmed Methods for General Purpose Digital Computers

The most common methods available to computer users are program library subroutines. The following examples are IBM oriented, but can be considered representative. Most of the coded subroutines available through the SHARE organization are for the square root only, and apply to floating-point operands. One of the fastest is SHARE distribution no. 721, which uses a least-squares approximation followed by two Newton-Raphson iterations, with a maximum relative error of 2.5×10^{-8}. The routine

requires 30 words of storage, and through clever coding executes a single-precision square root in 67 IBM 7090 machine cycles (1 cycle = 2.18 microseconds).

In contrast to the intricately coded case above, there is an n^{th} root subroutine (n integral) available (SHARE distribution no. 690) which builds up the root digit by digit in a trial-and-error fashion, checking each binary digit by raising the trial root to the n^{th} power, thus using a great many multiplications.

Lastly, it is interesting to note how the IBM FORTRAN II compiler sets up the exponentiation operation X**P. If P is an integer less than 8, the operation is executed as a series of P-1 multiplications. If P is an integer greater than or equal to 8, a log-exponential sequence is used. Also, if P is not an integer (as in the case of n^{th} roots), the log-exponential sequence is used. If the FORTRAN programmer desires the square root he may use the special routine (SQRT) provided, which uses the least-squares-two Newton-Raphson iteration sequence.

Objective and Scope

In this report five main classes of n^{th} rooting methods are discussed from the standpoint of timing and mechanization.

The first method, called the binomial theorem method, is in the same class as ordinary long division and is shown to be a higher-order extension of the division process. Its formulation relies heavily upon the values of the binomial coefficients for different values of n. Both restoring and nonrestoring methods are discussed, and a nonrestoring method using normalized remainders whose speed depends upon the statistical distribution of the various remainders during the rooting process is outlined. The simplest case, the square root, has been simulated and the resulting distribution of execution times obtained. Inherent difficulties in the binomial theorem method for higher roots are pointed out.

A second n^{th} rooting process considered is one that relies upon the operand being in a favorable interval such that its n^{th} root can be expressed as a correctable truncated series having very few terms. The operand is forced into this favorable interval by using stored constant multipliers obtained by table lookups. The nature of these constants as well as stored constants to correct the result obtained from the truncated series are presented, and table sizes are given as a function of speed

and accuracy. A related method which forces the operand into a given interval near unity while another transformation dependently forms the n^{th} root is discussed.

Another class of n^{th} rooting procedures covered are those derivable from Euler's formula. A derivation of m^{th} order n^{th} rooting processes obtainable from Euler's formula as developed by J. F. Traub in a recent article is presented and their timing and mechanization are discussed.

A fourth method considered is the approximation of the n^{th} root by a rational fraction which is the ratio of two polynomials involving the operand. This type of approximation is called the Padé approximation, after the mathematician who formulated it. A special case, the Padé approximation of order one, is analyzed in some detail with respect to its precision for different values of n.

Lastly, the familiar logarithm-antilogarithm method of extracting n^{th} roots will be treated, using as an example a configuration developed by Cantor, Estrin, and Turn which generates the elementary functions ln x and e^x for any given x.

For clearly competitive methods, comparisons are made with the log-exponential approach to the n^{th}

rooting problem, and the points at which mechanization of the methods in question become as time consuming as the log-exponential method are estimated. In all cases parameters such as hardware or storage requirements are defined along with the potential parallelism inherent in the procedure.

CHAPTER II

Application of the Binomial Theorem to the Extraction of Roots of Integral Order

A given positive real integer of nk digits may be represented in the usual positional notation as

$$A = D_{nk-1}B^{nk-1} + D_{nk-2}B^{nk-2} + \cdots + D_1 B + D_0 \,, \qquad (1)$$

where $\quad D_i = i^{th}$ digit, $0 \leq D_i < B$, and

B = base of the number system used.

Both n and k are positive integers, and thus A consists of an integral multiple of n digits. In addition, let it be required that

$$\sum_{j=1}^{n} D_{nk-j} > 0 \,, \qquad (2)$$

i.e., at least one of the n most significant digits of A is nonzero. Similarly, let another positive real integer of k digits and with the same base as A be given in positional notation as

$$a = d_{k-1}B^{k-1} + d_{k-2}B^{k-2} + \cdots + d_1 B + d_0 \,, \qquad (3)$$

where $\quad d_i = i^{th}$ digit, $0 \leq d_i < B$.

Let the two integers A and a be related by the reciprocal relations

$$a = \text{Int.}\{\alpha\} \quad \text{and} \tag{4}$$

$$A = \alpha^n , \tag{5}$$

where
$$\alpha = A^{1/n} , \tag{6}$$

and the operation Int.$\{ \}$ means the integer part of the expression in brackets. It is generally true that the positive real n^{th} root of a positive integer is not expressible exactly as another positive integer, and we shall regard $\underline{a}$ as the integer part of α , the exact positive real n^{th} root of A. The problem is, then, to determine the digits d_i of the integer part of the positive real n^{th} root of A having been given the digits D_i of A itself.

For convenience in notation, let us introduce the substitution

$$x_i = d_{i-1}B^{i-1} \tag{7}$$

into (3) in order that the expression for $\underline{a}$ assume a more convenient multinomial form. Doing this,

$$a = x_k + x_{k-1} + \cdots + x_1 \quad . \tag{8}$$

Now approximate α by its integer part, and substitute (8) into (5) yielding

$$A = (x_k + x_{k-1} + \cdots + x_1)^n \quad . \tag{9}$$

Let us now attack the problem in reverse fashion by focusing attention on the digits of $\underline{a}$. As a first approximation

let $a_1 = x_k$, i.e., let $\underline{a}$ be approximated by its highest order component[1]. In a like manner, then, a first approximation to A is defined as $A_1 = a_1^n = x_k^n$. Then let succeeding better approximations to $\underline{a}$ be defined as

$$a_j = \sum_{i=0}^{j-1} x_{k-i} \ , \ j = 1,2,3,\ldots \ , \qquad (10)$$

where $a_0 = 0$. Equation (10) clearly shows that $\underline{a}$ is being built up digit by digit toward the desired value, Int.$\{\alpha\}$. The j^{th} approximation to A is

$$A_j = a_j^n = \left\{ \sum_{i=0}^{j-1} x_{k-i} \right\}^n \quad . \qquad (11)$$

From equation (10) it is clear that

$$a_j = a_{j-1} + x_{k-j+1} \quad , \qquad (12)$$

and thus

$$A_j = (a_{j-1} + x_{k-j+1})^n \quad . \qquad (13)$$

Expanding (13) using the binomial theorem,

$$A_j = a_{j-1}^n + \left[n a_{j-1}^{n-1} x_{k-j+1} + \cdots + x_{k-j+1}^n \right]$$

or

$$A_j = A_{j-1} + \left[n a_{j-1}^{n-1} x_{k-j+1} + \cdots + x_{k-j+1}^n \right] \quad . \qquad (14)$$

By definition, $A_0 = 0$.

Equations (14) and (12) represent an iterative sequence that may be used to extract the positive real n^{th} root of a given positive real integer. Since the integer part of the desired root is built up digit by digit, the

[1]By a component is meant the digit times the power of B.

sequence of approximations obeys $a_{j-1} \geq a_j$, and therefore the approximations a_j approach $\underline{a}$ monotonically from below. Equation (10) ensures that $a_k = a$, and that

$$\lim_{j \to \infty} a_j = \alpha .$$

Thus, $\alpha - a_j \leq \epsilon$, $\epsilon \geq 0$, i.e., the error $\alpha - a_j$ may be made as small as desired by merely executing more stages of the iterative process (14). We may, then, extract the n^{th} root of A beyond its integer part to as many places as desired.

Specialization to a Restoring [10] Type Procedure for Obtaining the Square Root of a Real Integer

Let us rewrite (14) by considering the remainder at each stage of the iterative process. Let $R_i = A - A_i$ and make this substitution in equation (14), giving

$$R_j = R_{j-1} - \left\{ na_{j-1}^{n-1} x_{k-j+1} + \cdots + x_{k-j+1}^{n} \right\} , \qquad (15)$$

where $R_0 = A$. R_j is the remainder that results from the j^{th} stage of the process. The j^{th} remainder is obtained by subtracting the terms in brackets from the previous remainder, thus obtaining a root digit in the process. Because the components x_i are postulated to be the actual components of the integer part of the exact n^{th} root, it is clear that $0 \leq R_j \leq R_{j-1}$.

Relation to Division

It is instructive to point out the similarity between the rooting process outlined in (15) and the restoring type division process. Using the notation of (8), we may write out the division problem $U/V = W$, where U is the dividend, V the divisor, and W the quotient.

$$(u_p + u_{p-1} + \cdots + u_1)/(v_q + v_{q-1} + \cdots + v_1)$$
$$(w_{p-q} + w_{p-q-1} + \cdots + w_1) \qquad (16)$$

where p and q are positive integers, $p > q$. In a manner similar to that of the rooting process, the quotient W may be built up digit by digit in the following manner:

$$W_j = W_{j-1} + w_{p-q-j+1} \quad , W_0 = 0. \qquad (17)$$

Paralleling the rooting process, the j^{th} approximation to $U = VW$ may be written $U_j = VW_j$. Therefore,

$$U_j - U_{j-1} = V(W_j - W_{j-1}) = Vw_{p-q-j+1} \; . \qquad (18)$$

Introducing the remainder $R_i = U - U_i$, the division process (18) becomes

$$R_j = R_{j-1} - Vw_{p-q-j+1} \quad , \quad R_0 = U \quad , \qquad (19)$$

which displays its obvious similarity to the rooting process in (15). In fact, if $n = 1$ in (15), the rooting process reduces to the trivial division problem $A/1$ if the

process is carried out an infinite number of stages. It should be noted that the trial subtrahend $Vw_{p-q-j+1}$ in the division process (19) is functionally independent of the partial quotient W_{j-1}, whereas the trial factor $na_{j-1}^{n-1}x_{k-j+1} + \cdots + x_{k-j+1}^{n}$ in the rooting process (15) is functionally dependent on the partial root a_{j-1}. This dependence is linear in the case of the square root $(n = 2)$, quadratic in the case of the cube root $(n = 3)$, and so on. This functional dependence is important in the nonrestoring rooting process discussed later.

In order to mechanize the rooting process in (15) on electronic digital computing machinery, a simple systematic method for generating the trial factors is desired. Let us write (15) in the form $R_j = R_{j-1} - E_j^n(d)$, where $E_j^n(d) = na_{j-1}^{n-1}x_{k-j+1} + \cdots + x_{k-j+1}^{n}$. The argument d of $E_j^n(d)$ is the digit part of x_{k-j+1}, which is to be determined during the j^{th} stage of the process. Clearly $E_j^n(0) = 0$, so we need to know the B-1 trial factors $E_j^n(1)$, $E_j^n(2), \ldots, E_j^n(B-1)$. In the restoring method the trial factors are generally subtracted from the remainder in a "differential" fashion, i.e., $R_{j-1} - E_j^n(1)$, $R_{j-1} - E_j^n(2)$

$-\left\{E_j^n(2) - E_j^n(1)\right\}$, etc., until a negative remainder is sensed, at which time the process "regresses" one step by adding on the previously subtracted item. This approach obviously accomplishes the desired result, i.e., the smallest $R_{j-1} - E_j^n(d) \geq 0$ is computed, yielding the desired root digit d. If at any stage of the process the resulting remainder R_j is zero, the process terminates because an exact root has been found. The maximum length of a determines the maximum number of stages of the rooting process, since one root digit is obtained per stage. If the "differential" subtracting method is used, it is expected that on the average about $\frac{1}{2}(B-1) + 1$ subtractions plus one readdition must be performed per stage of the process. If the binary number system is used, the unknown root component x_{k-j+1} to be determined on the j^{th} stage may be assumed to have a digit part of "1", the trial factor $E_j^n(1)$ formed and compared with R_{j-1}, and the appropriate action taken.

Mechanization of the Binary Restoring Binomial Rooting Process

Assuming the above procedure,

$$x_{k-j+1} = 2^{k-j} \quad . \tag{20}$$

Substituting (20) into (15) gives

$$R_j = R_{j-1} - \left\{ 2^{k-j}\, na_{j-1}^{n-1} + \cdots + 2^{nk-nj} \right\}. \tag{21}$$

Because of the restriction placed upon A in (2), i.e., that at least one of the n highest order digits of A be nonzero, the highest order root digit must be nonzero. That is, $a_1 = 2^{k-1}$. Since $a_1 \leq a_2 \leq \cdots \leq a_{j-1} \leq a_j \leq \cdots \leq a_k$, then

$$2^{k-1} \leq a_{j-1} < 2^k \quad . \tag{22}$$

Let us now examine the mechanization required to execute each iterated stage of the process, i.e., generation of the trial factor for particular values of n, and subtraction from the remainder R_{j-1}. In the case of the square root (n=2),

$$R_j = R_{j-1} - \left\{ 2 \cdot 2^{k-j} a_{j-1} + 2^{2k-2j} \right\} . \tag{23}$$

Using (22),

$$2^{2k-j} \leq 2 \cdot 2^{k-j} a_{j-1} < 2^{2k-j-1} \quad . \tag{24}$$

Equation (24) shows that the highest order digit of the trial factor will always appear in bit position 2k-j at the beginning of the j^{th} stage of the process, which means that it moves one position right during execution of each stage of the iterative process. By noting that 2k-2j < 2k-j, j = 1,2,3,..., a "1" need only be inserted (not added) into bit position 2k-2j to account for the rest of the

trial factor, since a carry cannot occur because of (24). It is clear that the remainder R_j is decreasing in magnitude with each succeeding stage of the process. To economize on register requirements, let us shift the remainder left one bit position after the execution of each stage. This means that after j stages the remainder will be multiplied by 2^j. Inserting this in (23),

$$2^j R_j = 2^j R_{j-1} - \left\{ 2^{k+1} a_{j-1} + 2^{2k-j} \right\}, \qquad (25)$$

and thus the leading bit of the trial factor remains stationary throughout the entire square rooting process. A similar procedure can be applied to the expressions involved in the higher rooting processes. In the usual single precision case, a k-bit root is extracted from a k-bit operand, where k is the number of bits in a single precision word. If this is the case, the registers have the formats shown below:

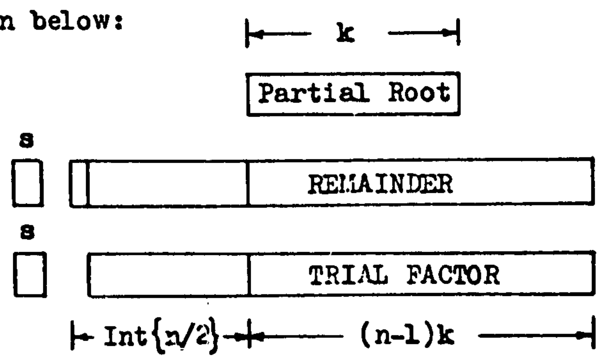


Figure 2-1: Register Formats for the Fixed-Point Binomial Theorem n^{th} Root.

The remainder register is $(n-1)k + \text{Int.}\{n/2\}$ bits, and the trial factor register is one bit less, or $(n-1)k + \text{Int.}\{n/2\} - 1$ bits, both registers having an additional sign bit. The partial root register must have attached to it some provision for building up the root bit by bit starting at the high order end. A counter with k sequential states and decoding circuits which select one input line at each stage of the process could enable this operation.

As n gets larger, the mechanization complexity increases. The additional terms acquired in the trial factor might be formed simultaneously in other registers or sequentially formed and added. For the case of extreme parallelism the extraction of the n^{th} root could utilize n-2 multipliers, n-2 shifters, and one adder in addition to the registers already mentioned.

Normalized Remainders

Recalling for the moment the square root algorithm in (25), we see that the trial factor is at least as large as 2^{2k}. It is then clear that if the previous remainder $R_{j-1} \leq 2^{2k-1}$, i.e., it has "leading" zeros, R_{j-1} may be shifted left until a "1" appears in bit position 2k-1. As a result, additional zero bits are introduced into the

partial root, one for each position the remainder is shifted left. The advantage of this procedure is that additional digits of the root are generated using simple shifts, without having to resort to time consuming comparisons. The number of normalizing shifts made at any given point in the iterative process depends upon the statistical distribution of the remainder magnitude throughout the rooting process. Following C, V. Freiman [4] , let us establish a "figure of merit" for the restoring algorithm with normalized remainders by defining an iteration as a comparison and conditional subtraction, a normalization, formation of a new trial factor, and conditional alteration of the partial root. Thus it is seen that an iteration may consist of more than one stage of the rooting process. The figure of merit is the number of root bits formed during each iteration. Similar remainder normalization procedures may be defined for the higher order rooting processes.

Nonrestoring [10] Algorithm for n^{th} Rooting

The binary rooting methods previously discussed were of the restoring type. As is done in division, a nonrestoring modification of the restoring procedure may be employed to extract the n^{th} root of a binary integer.

Suppose, on each stage of the process, the digit part of the desired root component x_{k-j+1} is assumed to be a "1" as was done in the restoring procedure. Let the trial factor be formed as usual, but now let negative remainders be allowed. Let us now proceed in such a way as to decrease the magnitude of the remainder, i.e., when $R_{j-1} > 0$ <u>subtract</u> the trial factor from it; when $R_{j-1} < 0$ <u>add</u> the trial factor to the remainder. Provided the root digits are formed correctly, using the nonrestoring scheme ought to offer a time advantage over the restoring method, because addition or subtraction of the trial factor takes place without regard to the relative magnitudes of the remainder and trial factor (assuming all normalizing shifts have taken place), but only with regard to the <u>sign</u> of the remainder R_{j-1}.

<u>Nonrestoring n^{th} Rooting Method With Normalized Remainders</u>

As was the case in the restoring n^{th} rooting algorithm, the trial factor has a fixed minimum magnitude. Thus, by noting the magnitude of R_{j-1}, normalizing shifts can be made to introduce additional digits into the partial root without the necessity of addition or subtraction. The process is uncomplicated if we consider a signed magnitude number representation.

Suppose we are in the j^{th} stage of the rooting process, the remainder is positive and normalized, and the trial factor has been formed. The difference is then formed, and let us suppose that this resulting difference is negative. Intuitively, by a comparison to the restoring method we know that the digit part of x_{k-j+1} has been found to be zero, so let the partial root be augmented with this zero bit. Now the new (negative) remainder, adjusted left one bit position to account for the factor 2^j, may or may not have leading zeros with respect to the fixed minimum magnitude of the next trial factor. If the remainder does not have any leading zeros, the new trial factor is formed and added to the (negative) remainder. If the new remainder has leading zeros, certain difficulties arise. The nonrestoring division process parallels its restoring counterpart in that the remainders, except for position relative to an arbitrary fixed reference, are the same at those points where the remainder changes sign from negative to positive in the nonrestoring process. However, the trial factors in the rooting processes are functionally dependent upon the partial root, and therefore the remainders in the restoring and nonrestoring algorithms will not correspond unless some sort of correction is

made. Such correspondence to the restoring algorithm is sufficient to guarantee that the correct n^{th} root is extracted. Thus, when the trial factor is added to a negative remainder, a correction is also added. The negative remainder's leading zeros are shifted out in a manner similar to that when the remainder is positive, except that in order to ensure that the remainder changes sign from negative to positive, it is shifted left until a "1" appears in the bit position directly to the right of the highest order bit position of the trial factor. However, when the remainder is negative, 1's are introduced into the partial root for every bit position that the remainder is shifted left. Again, it is seen that this corresponds exactly to what would occur given the same remainders at the beginning of the stages involved in the remainder's changes of sign and normalization.

To illustrate the mechanics of this process, an example of the restoring and nonrestoring methods applied to a binary square root is given in Figure 2-2. Assume we are in the interior of a square rooting process, and the remainder is 0.101011101, the trial factor is 0.1011101, and the partial root is 0.10111 . The symbols are R = remainder, TF = trial factor, and C = correction.

RESTORING

	Registers	Partial Root
R	+0.101011101	0.10111
TF	-0.1011101	
R	+0.101011101	0.101110
TF	-0.010111001	
R	+0.010100100	0.1011101
TF	-0.00101110101	
R	+0.00100011011	0.10111011
TF	-0.0001011101101	
R	+0.0000101111111	0.101110111

NONRESTORING

	Registers	Partial Root
R	+0.101011101	0.10111
TF	-0.1011101	
2R	-0.00010111	0.101110
TF	-0.Shift	
8R	-0.010111	0.10111011
TF	+0.1011101101	
16R	+0.101111101	
C	+0.000000010	
16R	+0.101111111	0.101110111

Figure 2-2: Correspondence Between Restoring and Nonrestoring Square Root Processes.

Corrections to Remainders in the Binary Nonrestoring Rooting Process

It is expected that the correction that must be made to some of the remainders during the nonrestoring rooting process will depend upon both the partial root and the number of shifts required to normalize the remainder. To determine the value of the correction, the re-

storing and nonrestoring versions of a given iteration will be compared, and the difference in the final remainders will be the desired correction. Let us therefore consider a group of stages of the nonrestoring process which consists of one subtraction to get a negative remainder, a normalizing shift of s bit positions, and one addition that again yields a positive remainder, and compare those factors which are subtracted from the remainder R_{j-1} with the corresponding factors in the restoring process. Let us consider the square root process first.

A. Restoring Method:

$$F_s^R = -\left\{2a_j\, 2^{k-j-1} + (2^{k-j-1})^2\right\} - \left\{2a_{j+1}\, 2^{k-j-2} + (2^{k-j-2})^2\right\} - \cdots - \left\{2a_{j+s}\, 2^{k-j-s-1} + (2^{k-j-s-1})^2\right\} \tag{26}$$

The relation between successive partial roots is

$$a_{j+s} = a_j + \sum_{i=0}^{s-1} 2^{k-j-i-1}, \quad 0 \le s \le k-1.$$

Then

$$F_s^R = -2^{k-j}\left\{2a_j(1-2^{-s-1}) + 2^{k-j}\,(1-2^{-s}+2^{-2s-2})\right\} \tag{27}$$

B. Nonrestoring Method:

$$F_s^{NR} = -\left\{2a_{j-1}\, 2^{k-j} + (2^{k-j})^2\right\} - \left\{2a_{j+s}\, 2^{k-j-s-1} + (2^{k-j-s-1})^2\right\}$$

Since $a_{j-1} = a_j$,

$$F_s^{NR} = -2^{k-j}\left\{2a_j(1-2^{-s-1}) - 2\ 2^{k-j}(1-2^{-s})\ 2^{-s-1} + 2^{k-j}(1-2^{-2s-2})\right\} \tag{28}$$

Taking the difference between (27) and (28),

$$F_s^{NR} - F_s^{R} = -2^{2k-2j}\ 2^{-2s-1} \quad . \tag{29}$$

As was expected, equation (29) indicates that too much was subtracted from the remainder R_{j-1}, and thus the indicated correction must be added to the normalized negative remainder along with the new trial factor in order to achieve the desired relation $F_s^{NR} - F_s^{R} = 0$. In order to transform the correction in (29) to a value applicable to the modified algorithm of equation (25), it must be multiplied by 2^{j+s+2}, because the process has advanced j+s+2 stages since its beginning. Thus,

$$C_2^s = -2^{j+s+2}(F_s^{NR} - F_s^{R}) = 2^{2k-j-s+1},\ 0 \leq s \leq k-1, \tag{30}$$

where C_2^s is the correction that must be added to the normalized negative remainder along with the new trial factor after a normalizing shift of length s, for the nonrestoring binary square root $(n=2)$ process with normalized remainders.

It has turned out that the remainder correction for the square root process is dependent only upon a

single bit position, and not upon the partial root. However, a short examination reveals that the correction is more complex for the higher rooting processes. For the square root the correction is a zeroeth order polynomial in the partial root, for the cube root a first order polynomial in a_{j-1}, and so on.

<u>Extensions of the Method to Floating-Point Operands</u>

The binomial theorem method developed so far has been used for extracting the integral roots of binary integers, and is naturally extendable to fixed-point numbers of finite but variable precision, since the only difference between the two is the arbitrary placement of the binary point. The method may be easily extended to compute the roots of floating-point operands, i.e., a mantissa part multiplied by a power of the radix, by altering the mantissa (or fraction) according to the radix exponent. Specifically, let us consider binary floating-point operands of the form $A = f \cdot 2^b$, where $1/2 \leq f < 1$, i.e., the operand A has a normalized fractional part f. Let us now examine the exponent b. When taking the n^{th} root of $f \cdot 2^b$, we must form b/n, desiring this division to have a zero remainder. Suppose $b/n = \text{Int.}\{b/n\} + r/n$. Then if we take

$$A = 2^{-(n-r)} f \cdot 2^b = f' \cdot 2^{b'} \quad ,$$

where $b' = b+n-r$, $0 \leq r < n$, the desired rooting can be done. Since $2^{-1} \leq f < 1$, the altered fraction will lie in the range $2^{-(n-r+1)} \leq f' < 2^{-(n-r)}$, which still satisfies equation (2).

Additional Mechanization Requirements for the Nonrestoring Method

In general, scientific-type computations make extensive use of the floating-point representation. Therefore, because there is the possibility of shifting the operand fraction as many as n-1 positions to the right before performing the n^{th} root, this number of positions must be added onto the low order end of the remainder and trial factor registers, in order to retain a precision of 1 part in 2^k when extracting a k-bit root.

An additional set of registers must be provided for the formation of the remainder correction, which is a polynomial of order n-2 in the partial root a_{j-1}. If extreme parallelism is used, the extraction of the n^{th} root could utilize the partial root, remainder, trial factor, and correction registers, and 2n-3 multipliers, 2n-4 shifters, and 2 adders.

CHAPTER III

Design and Simulation of a Binary Square Root Device Employing the Binomial Theorem Method

The fixed-point nonrestoring binary square root algorithm given in equations (2-25) and (2-30) may be mechanized as a digital macro-operation in much the same manner as division. For the sake of reference, the algorithm equations are reproduced below for the remainder at the j^{th} iteration:

$$2^j R_j = 2^j R_{j-1} - 2^k \left\{ 2a_{j-1} + 2^{k-j} \right\}, \; j = 1,2,\ldots,k, \quad (1)$$

where $R_0 = A$, and the post-normalizing correction is

$$C_2^s = 2^{2k-j-s+1}, \; 0 \leq s \leq k-1 \; . \quad (2)$$

Let us consider the binary operands as being in the form

$$A = 2^E \cdot f \; , \quad (3)$$

where $1/2 \leq f < 1$, and E has positive or negative values. As a particular example, let the floating-point binary operand in (3) be of the form used in the IBM 7090, namely, a 27-bit fractional part, an 8-bit characteristic, and a sign bit, making up a 36-bit binary word. In the IBM floating-point format, the characteristic is formed by adding 128 to the exponent E, thus disallowing negative characteristics and restricting the exponent range to

(-127, 127). Negative exponents, then, are represented symbolically by characteristics in the range (1, 127). Extraction of the square root of such an operand will be

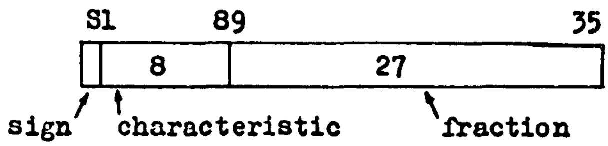


Figure 3-1: IBM 7090 Floating-Point Binary Format.

achieved by performing a fixed-point binary square root upon the fraction part, and halving the characteristic. However, there are two cases which must be considered.

<u>Case 1: E Odd</u>

If the exponent E and therefore the characteristic of the operand is odd, the fraction part f must be multiplied by 1/2 (shifted right one bit position) and the fixed-point square rooting process initiated. The characteristic of the resulting floating-point square root is formed by halving the operand characteristic, adding one to the units position (bit 8), and then adding 64 to the result to form the correct value. The above method is formulated as

$$(2^{E}\cdot f)^{1/2} = 2^{\mathrm{Int.}\{\frac{1}{2}E\}+1}\cdot(\tfrac{1}{2}f)^{1/2} \quad . \qquad (4)$$

Since $1/2 \leq f < 1$, then $1/4 \leq \frac{1}{2}f < 1/2$, and so

$1/2 \le (\frac{1}{2}f)^{\frac{1}{2}} < 1/\sqrt{2}$; thus the fraction part of the square root is normalized. The characteristic of the square root is formed according to

$$\text{Int.}\left\{\tfrac{1}{2}(E + 128)\right\} + 1 + 64 = (\text{Int.}\left\{\tfrac{1}{2}E\right\} + 1) + 128. \quad (5)$$

Case 2: E even

If the operand characteristic is even, i.e., it has a zero in its units position, then the characteristic is simply halved and 64 added to it, and the fixed-point binary square rooting process is applied to the unmodified fraction part, f. Symbolically,

$$(2^{E}\cdot f)^{1/2} = 2^{\frac{1}{2}E}\cdot f^{1/2} \text{ , and} \quad (6)$$

$$\tfrac{1}{2}(E + 128) + 64 = \tfrac{1}{2}E + 128 \quad . \quad (7)$$

A straightforward magnitude analysis of the remainders in the rooting algorithm (1) shows that if the initial remainder R_0 (which is the fractional part of the operand itself) is inserted into a 27-bit register, an extra bit position to the right of the 27 bits is needed in order to save the lowest-order bit of the operand. This will make the remainder register a total of 29 bits plus sign, and the trial factor register has one less bit, or a total of 28 bits plus sign. Now let us combine the remainder and trial factor registers into a binary accumulator, the remainder register being the accumulator register, and the

trial factor register being the addend or subtrahend register, depending upon whether the accumulator is the adding or subtracting type. An examination of the additive/subtractive processes during the square rooting procedure reveals that only three cases are allowed:

1). $R^{+} - TF^{+} \geq 0$ R^{-} = positive remainder

2). $R^{+} - TF^{+} < 0$ R^{-} = negative remainder

3). $R^{-} + TF^{+} > 0$ TF^{-} = positive trial factor

If the accumulator is made a binary subtracting accumulator (with an accumulator and subtrahend register), then $C(AC) = C(AC) - C(SU)$ represents its operation symbolically. Further, let negative numbers be represented in 1's complement form, and let the sign bit be 0 for positive, 1 for negative. In this case the three cases become

Case	end-around borrow?
1). $R^{+} - TF^{+} \geq 0$	no
2). $R^{+} - TF^{+} < 0$	yes
3). $R^{-} - (-TF^{+}) > 0$	no

For each case 2 that occurs it is expected that a case 3 will subsequently occur, unless the rooting process is terminated during the normalizing shift or before the normalizing shift takes place. In case 3 the term $-TF^{+}$ is

represented as a 1's complement. In the 1's complement representation of negative numbers, the complement digits are just the inverse of the digits in the true representation, and thus leading zeros in the true representation are leading ones in the complement representation. Therefore normalization of the remainder takes place either with a zero (+) sign bit and leading zeros, or a "1" (-) sign bit and leading 1's, zeros augmenting the partial root in the former case, and 1's in the latter. A characteristic of the 1's complement representation is the occurrence of an end-around borrow (or carry) as in case 2. Using suitable borrow look-ahead circuitry (such as in the IBM 7090), the end-around borrow may be reckoned along with the normal borrows that occur. Thus, subtraction takes a fixed minimum time, whether the end-around borrow occurs or not. Note that there is no ambiguity in the representation of the quantity "zero", since only -0 occurs (case 1).

Let us assume that our accumulator automatically adjusts the final difference left one bit position upon the execution of each subtraction to account for the factor 2^j in the algorithm (1). The accumulator register must be equipped to shift left or right one bit position upon

the reception of left shift or right shift signals, zeros being introduced into the positions vacated. When the normalized remainder is negative, both the 1's complement of the new trial factor and the 1's complement of the correction must be subtracted from it. The only other operations to be considered in the fixed-point square root are the augmenting of the partial root, formation of the new trial factor from the partial root, and the formation of the remainder correction bit. Because of the simple relationship between the trial factor and the partial root (eqn.(1)), there is no necessity to carry the partial root in a separate register, since it can be clearly identified as an extractable part of the trial factor, and extracted from the trial factor register at the end of the rooting process. The organization of the fixed-point square rooter is given in Figure 3-2. The logical equations for the various control signals emanating from the local control are given later in this chapter. The local control directs the rooting process according to the various decisions that have to be made. A flow chart describing the square rooting sequence and the inherent decisions involved is given in Figure 3-3. In the flow chart, the following symbols are used:

DGLINE = digit line selector;

TFR = trial factor register;

REMR = remainder register.

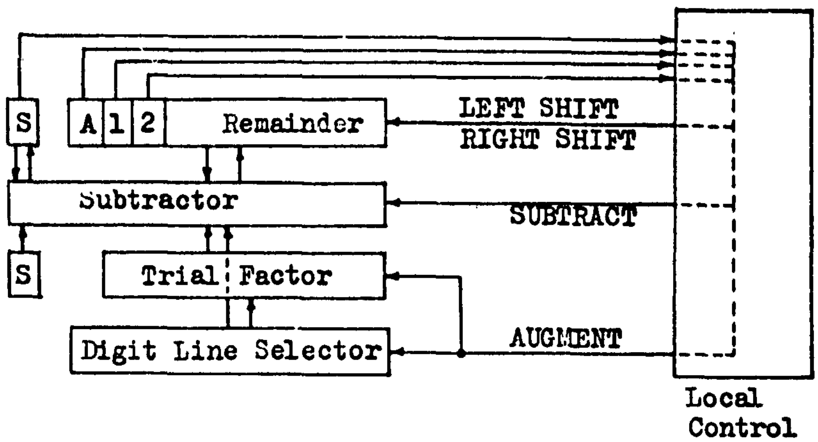


Figure 3-2: Organization of the Fixed-Point Square Rooter.

It has been shown that when the remainder becomes negative in the nonrestoring rooting process, a correction must be added to the remainder along with the next trial factor. Specifically, the post-normalizing correction for the square root is given in equation (2) as $C_2^s = 2^{2k-j-s+1}$, $0 \leq s \leq k-1$, where s̲ is the number of normalizing shifts made during the iteration in question. An examination of the above expression reveals that it is exactly the bit position corresponding to the digit line that is enabled at the time that the addition of the trial factor and the

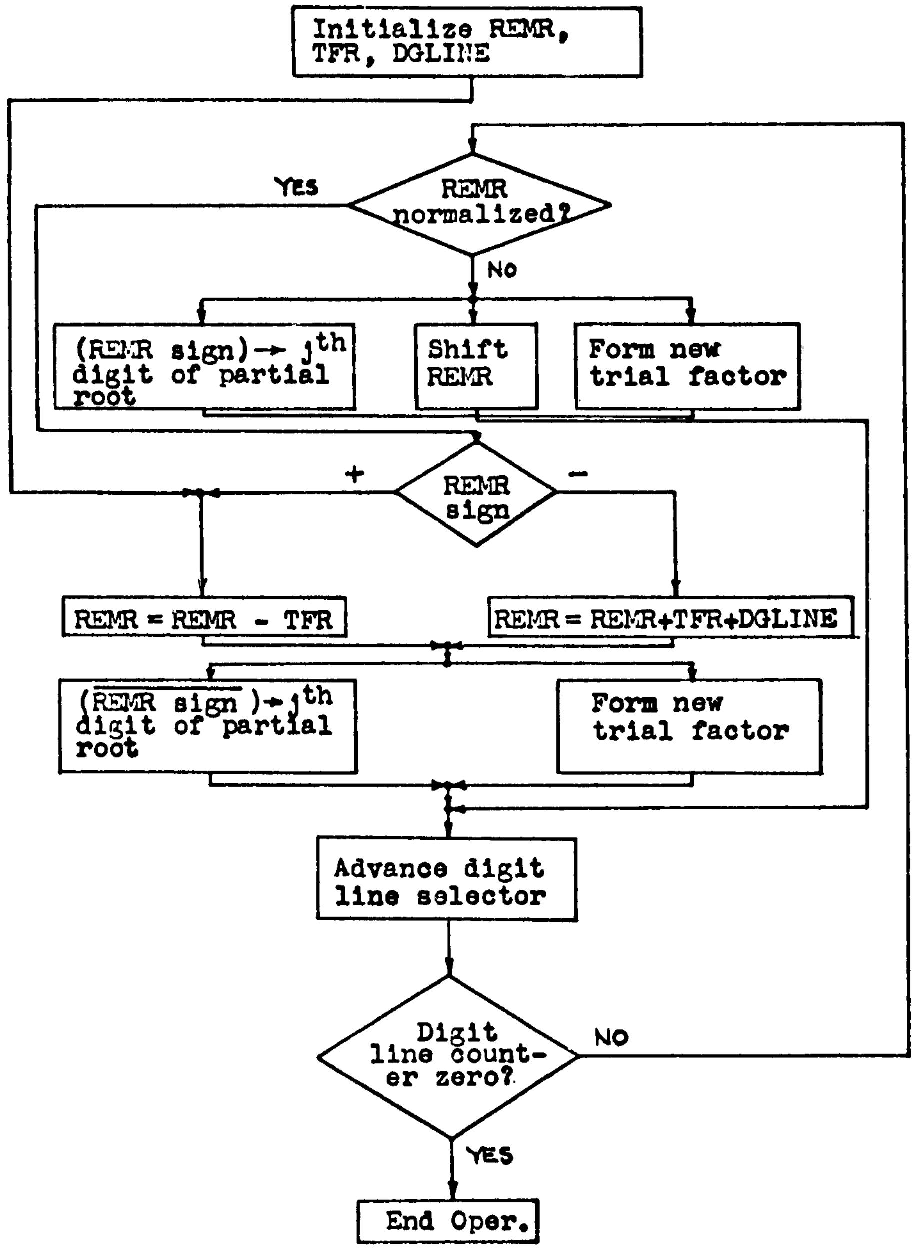


Fig. 3-3: Binary Square Root Micro Flow Chart Mantissa Part.

(negative) remainder takes place. Therefore it is possible to consider mechanization of the subtraction and correction functions in parallel, with the addition of the digit lines being suppressed when the remainder is positive, i.e., when its sign bit is a zero.

Internal States and Control Logic for the Fixed-Point Square Rooter

The operation of the binary square rooter may be given in a state table which describes the sequential computation in terms of the states of a counter. The state table is given in Table 3-1. The three basic operations in the fixed-point part of the binary square root are subtraction of the trial factor from the remainder, augmenting the partial root after the subtraction, and simultaneously shifting out leading zeros and further augmenting the partial root. The basic decisions made during the process depend upon the disposition of the remainder, trial factor, digit line selector, and the state counter. The state counter counts in the sequence given in Table 3-1. There is another counter, the digit line counter, that changes state every time a different digit line is to be enabled. This counter has 27 states, and thus requires 5 memory elements. We shall let the counter be 26 $(11010)_2$

State Counter T1 T2		Operation
0 0		Examine REMR S,A,1,2 and position 1 of DGLINE selector.
	$(\overline{S})(\overline{A})(\overline{1})(\overline{\mathrm{DGLINE1}}) + (S)(A)(1)(2)(\overline{\mathrm{DGLINE1}})$	Shift REMR A, 1-29 left one position. Simultaneously present AUGMENT signal. Advance to state 10.
	$(\overline{S})\{\overline{(\overline{A})(\overline{1})}\} + (S)\{\overline{(A)(1)(2)}\}$	Perform subtraction. If S = 0 REMR = REMR - TFR. If S = 1, REMR = REMR - comp.(TFR) - comp.(DGLINE). Advance to state 11.
1 1		Form AUGMENT signal. Advance to state 10.
1 0		Advance digit line selector. Advance to state 01.
0 1		Examine digit line counter: $\neq 0$: Advance to state 00; $= 0$: End operation.

Table 3-1: Table of Basic States for the Execution of the Fixed-Point Part of the Binary Square Root.

to enable DGLINE 1, and zero (00000) to enable DGLINE 27, the intervening states being assigned in descending order. When DGLINE 1 = 1, the possible shifting out of a leading zero is suppressed (state 00). The important register bit positions are the remainder S, A, 1, 2, as shown in Figure 3-2. The remainder left shift one bit-position signals are derived as follows:

1). Remainder Positive (S = 0):

$$\text{LEFT SHIFT} = (\overline{\text{S}})(\overline{\text{A}})(\overline{1})(\overline{\text{DGLINE 1}})(\overline{\text{T1}})(\overline{\text{T2}})$$

$$\text{SUBTRACT} = (\overline{\text{S}})\{\overline{(\overline{\text{A}})(\overline{1})}\} + (\text{DGLINE 1})(\overline{\text{T1}})(\overline{\text{T2}})$$

2). Remainder Negative (S = 1):

$$\text{LEFT SHIFT} = (\text{S})(\text{A})(1)(2)(\overline{\text{DGLINE 1}})(\overline{\text{T1}})(\overline{\text{T2}})$$

$$\text{SUBTRACT} = (\text{S})\{\overline{(\text{A})(1)(2)}\} + (\text{DGLINE 1})(\overline{\text{T1}})(\overline{\text{T2}})$$

The AUGMENT signal is generated during states 00 and 11, and is derived from the following:

$$\text{AUGMENT} = (\overline{\text{DGLINE 1}})\,\{(\overline{\text{S}})(\overline{\text{A}})(\overline{1}) + (\text{S})(\text{A})(1)(2)\}(\overline{\text{T1}})(\overline{\text{T2}}) + (\text{T1})(\text{T2}) \,.$$

The digit line counter may be counted down one step upon the reception of the AUGMENT signal, provided that there is a delay in the change of state so that the original state of the counter may be interrogated.

Recalling that the trial factor is given by

$2^k(2a_{j-1} + 2^{k-j})$ in equation (1), its format at a given stage is .XX···XX01, where the X's (j-1 of them at the j^{th} stage) represent $2^k 2a_{j-1}$, the "0" represents the current root bit which is to be determined, and the "1" is the term $2^k 2^{k-j}$. During the next stage, i.e., the $(j+1)^{st}$, the trial factor has j X's followed by a zero and a one. Thus, augmenting the partial root and forming the next trial factor may be done at the same time in a single logical operation, as illustrated in Figure 3-4. The logical operations of augmenting the partial root (contained in TFR) and forming the new trial factor are given by the following equations:

$$
\begin{aligned}
{}_{S}TFR_i &= (AUGMENT)\{(\overline{S})(DGLINE)_i(T2) - (S)(DGLINE)_i(\overline{T2}) \\
&\qquad (DGLINE)_{i-2}\} \\
{}_{R}TFR_i &= (AUGMENT)(DGLINE)_{i-1}
\end{aligned}
$$

<u>Timing Study of the Square Root Device</u>

Since the execution time of the square rooting device depends upon the statistically distributed magnitudes of the intermediate remainders in the square rooting process, it is expected that the execution time itself will possess some sort of statistical distribution. This distribution is very difficult to obtain by any method other

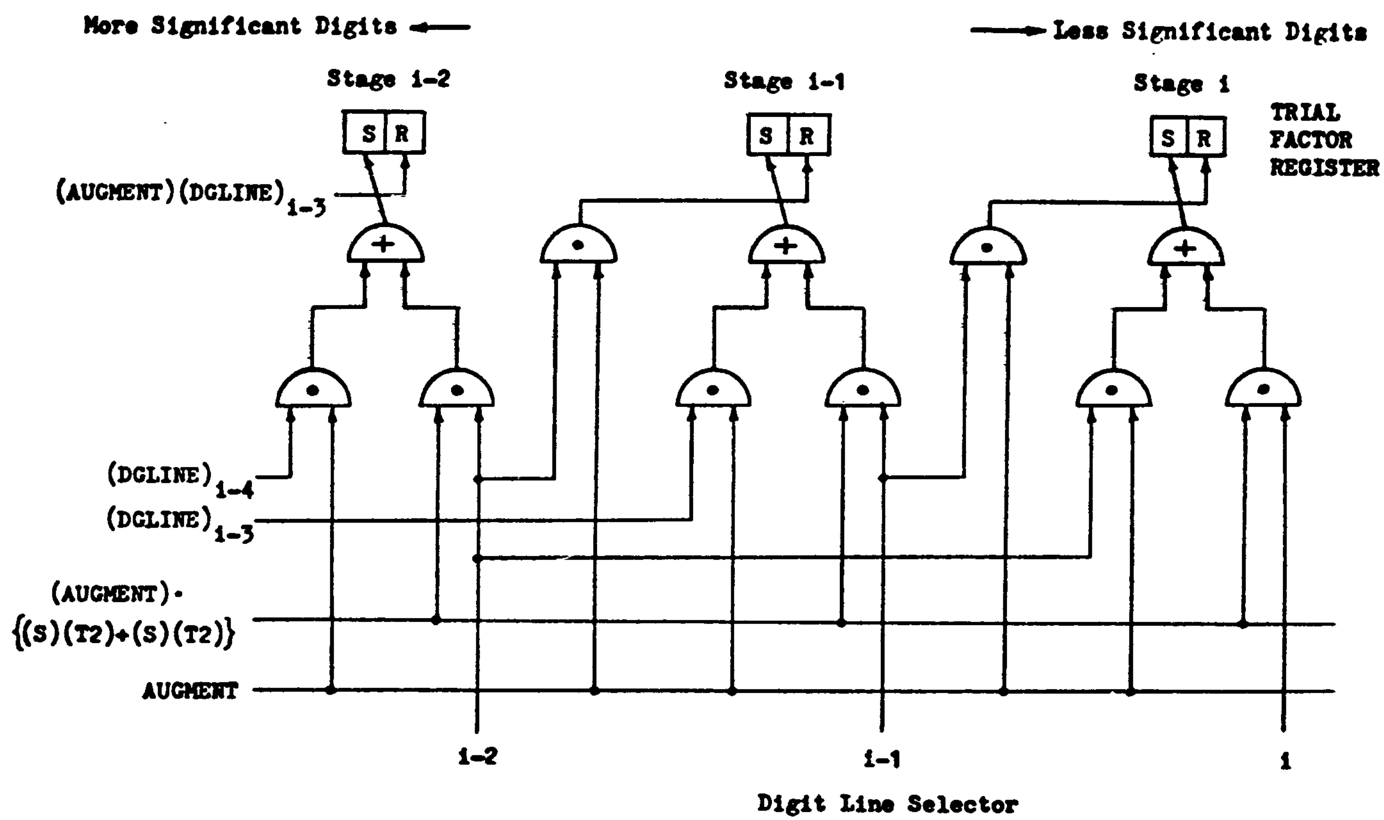


Figure 3-4: Simultaneously Augmenting the Partial Root and Forming the Trial Factor.

than direct experimental simulation, since the distribution of the remainder magnitudes depends upon the previous remainders and the partial root during the square rooting process. A computer program for the IBM 7090 was written to simulate the operation of the square rooter, thereby enabling certain characteristics of the method to be determined. The basic format of the numerical experiments performed is shown below:

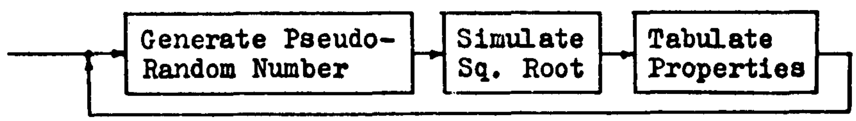
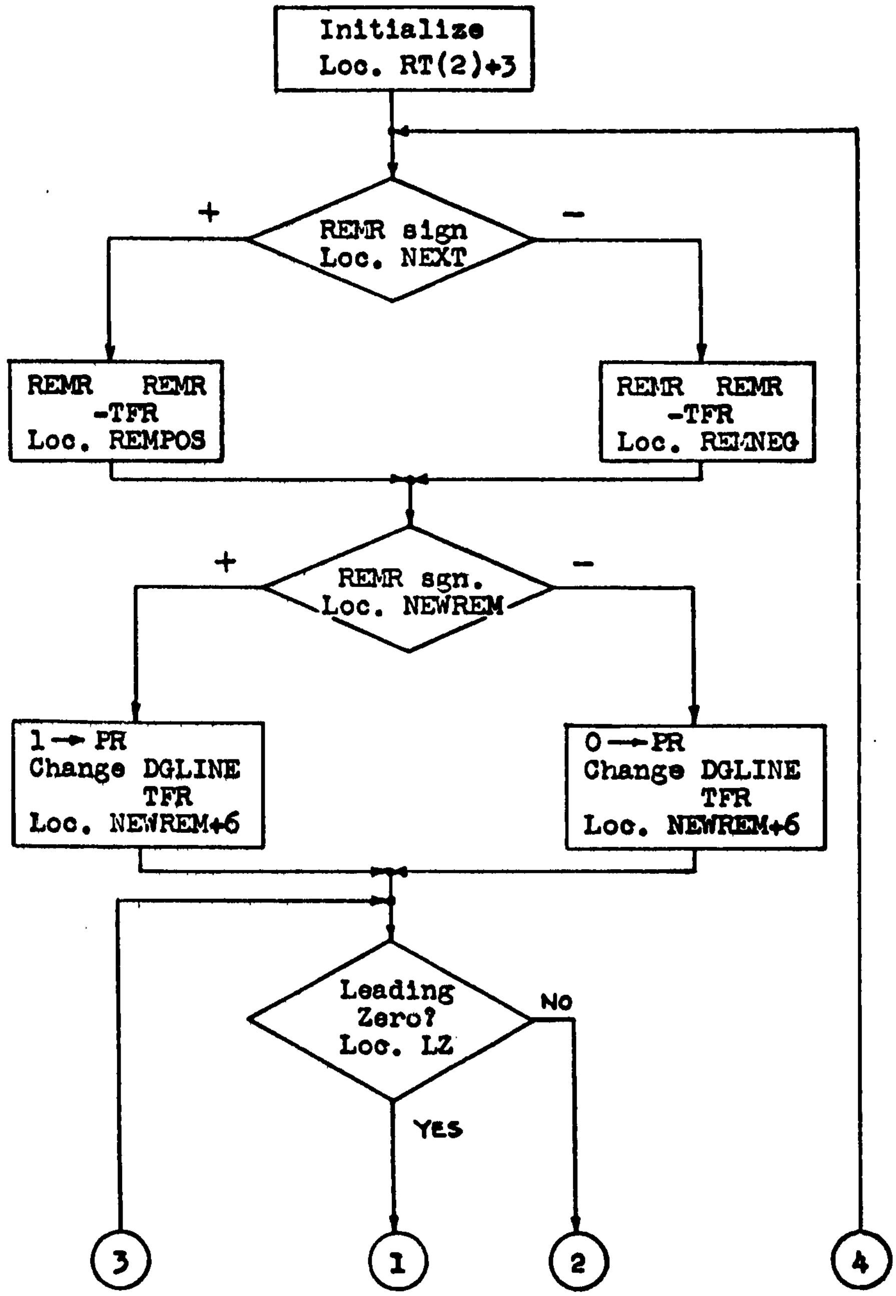


Figure 3-5: Basic Format of Numerical Experiments.

The simulation experiments were performed upon the fraction part of an IBM floating-point operand, since this is the part of the process which is of major interest, and in fact is the dominant factor in the execution time. The fraction parts of the floating-point words were in the interval (1/4, 1), but were generated in the interval (1/2, 1) by a pseudo-random number generator. A flow chart of the binary square root simulation program is given in Figure 3-6. The symbolic locations given in the flow chart correspond to the locations in the program listing (see Appendix) at which the indicated operations occur.


Initialize
Loc. RT(2)+3
REMR sign
Loc. NEXT
+
−
REMR REMR
-TFR
Loc. REMPOS
REMR REMR
-TFR
Loc. REMNEG
REMR sgn.
Loc. NEWREM
+
−
1 → PR
Change DGLINE
TFR
Loc. NEWREM+6
0 → PR
Change DGLINE
TFR
Loc. NEWREM+6
Leading
Zero?
Loc. LZ
NO
YES
3
1
2
4

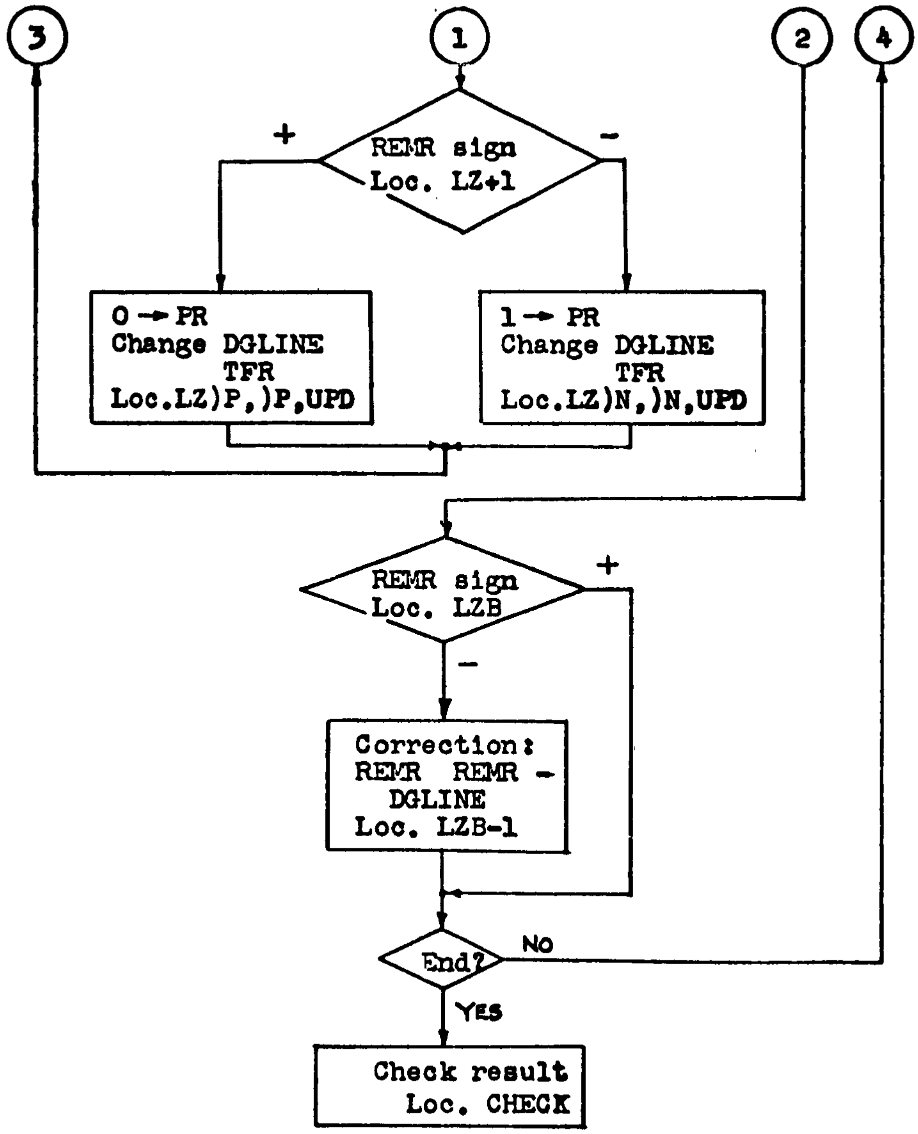


Fig. 3-6: Flow Chart for Binary Square Root Simulation Program, Mantissa Part, pp. 123-129 , 143-148 .

Pseudo-Random Number Generator

The pseudo-random number generator used in the numerical experiments was a multiplicative congruential type as described by Rotenberg [11]. The multiplicative congruence algorithm is

$$x_{i+1} = (2^a + 1)x_i + C \text{ , Mod. } 2^p \text{ ,} \tag{8}$$

where a is a real integer. Rotenberg applied several empirical tests to the above algorithm with $a = 7$, $C = 1$, and $p = 35$. He found that the resulting numbers were uniformly distributed and that there was no detectable serial correlation in the sequence. The cycle structure of the multiplicative congruence method has been determined analytically, and it is known that algorithm (8) can generate the full period of 2^p numbers if $a = 2$ and C is odd [11]. The serial correlation between two consecutive numbers in the sequence has been shown by Coveyou [3] to be

$$\rho(x_i, x_{i+1}) = \frac{1 - 6C \cdot 2^{-p}(1 - C \cdot 2^{-p})}{2^a + 1} \text{ .} \tag{9}$$

The 27-bit pseudo-random numbers used were generated in the interval (1/2, 1) by first generating a 26-bit pseudo-random number, and then putting a "1" in front of it, making a 27-bit number. The algorithm parameters used in (8) were $a = 11$, $C = 1$, and $p = 26$, and the resulting serial cor-

relation between two successive numbers is, from (9),

$$\rho(x_i, x_{i+1}) = \frac{1 - 6\cdot 2^{-26}(1 - 2^{-26})}{2^{11} + 1} \approx 0.0005 .$$

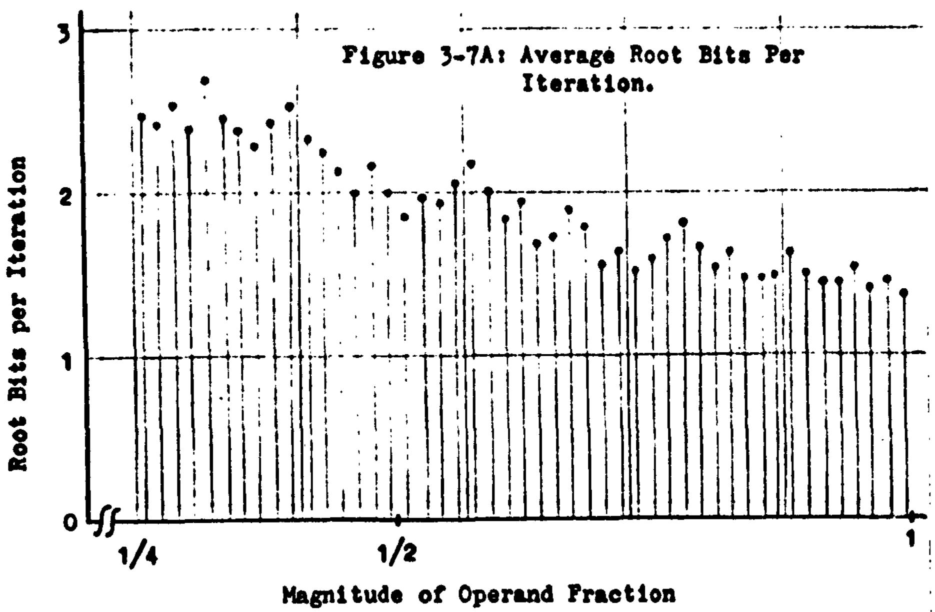
The initial random number x_0, in octal form, was 232544614, but other runs of the experiment showed, as should be the case, that the results were insensitive to x_0 after a reasonable sequence length in (8).

Experiment I: Property Distribution

To reveal in a general way the efficiency of the nonrestoring square root method with normalized remainders, the previously defined figure of merit "root bits per iteration" was obtained as a function of the magnitude of the operand characteristic. No knowledge was assumed concerning the nature of the operands, other than that they belonged to the class of all properly normalized binary floating-point operands of the IBM format. Therefore it was assumed that the operand fractions were uniformly distributed over the interval (1/4, 1). If something more were known about the nature of the operands, it might be possible to restrict the interval of interest, and in general entirely different conclusions concerning the method's computational efficiency relative to the subinterval of interest could be drawn. As an additional point of int-

erest, the average number of corrections per operand (27-bit fraction) was also determined, and plotted versus the fraction part. For the experiment, the interval (1/4, 1) was subdivided into 48 parts, making the class interval equal to 1/64. The results were averaged within each interval, since only the trend of the properties in question was desired.

The results are shown in Figure 3-7. It is apparent that there is a general decrease in efficiency and hence an increase in execution time as the magnitude of the operand fraction increases, since there is a decreasing number of root bits per iteration being obtained, as shown in Figure 3-7A. The irregularities in the curve are due to the dependence of the method's speed upon the patterns of ones and zeros in the root itself, and thus are difficult to trace back to the bit arrangements in the operand. However, there is a definite trend shown, and the minimum average root bits per iteration obtained was 1.38 in the subinterval (63/64, 1), the maximum was 2.70 in the subinterval (5/16, 21/64), and the mean value was 1.91 root bits per iteration in the entire interval. The minimum and maximum given, of course, are not absolute, since averaging the results in each class interval "blunted" these



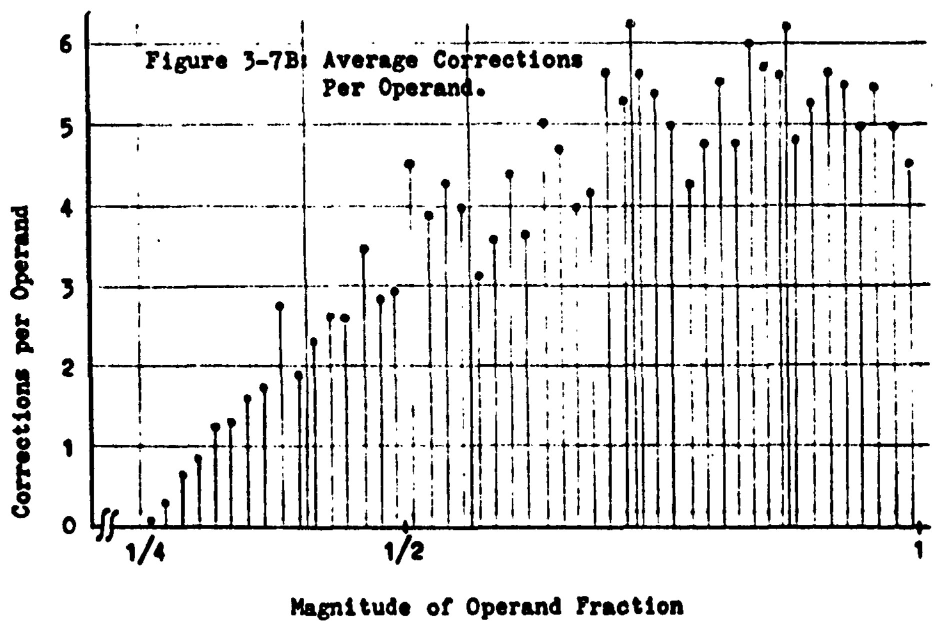


Figure 3-7: Properties of the Nonrestoring Square Root Method Using Normalized Remainders.

values. Thus, in taking the square root of the fraction part of a normalized floating-point binary number drawn at random from the population of all numbers of this type, the expected figure of merit is about 1.91 root bit per iteration, i.e., it is expected that an average of 0.91 root bits will be obtained by normalizing the remainder each iteration.

In the development of the nonrestoring binomial theorem method it was shown that the remainder must be corrected each time it becomes negative. To get an idea of how many times this occurs on the average per operand, the average number of corrections per operand was measured in the same way as the number of root bits per iteration was. The results are given in Figure 3-7B. The measured average minimum was about 0.05 corrections per operand, the maximum about 6.03, and the mean about 3.85.

Experiment II: Timing Distribution

In order to evaluate the performance of the binomial theorem square rooting method with respect to execution time, another numerical experiment was performed, and this time the total execution time taken to operate upon a floating-point binary operand was measured in terms of a defined time unit. The previously discussed device using

the 1's complement representation for negative numbers was investigated as a particular example. Throughout the square root process there are certain time costs which must be "paid" in order to accomplish the various functions involved. These time costs represent different phases of the process, and were chosen as modifiable parameters which influenced the total execution time of the process in varying degrees. The following parameters were chosen:

1). T_{add} = time taken to execute the subtraction of the trial factor from the remainder;

2). T_a = time taken to augment the partial root and form the new trial factor; and

3). T_s = time taken to shift the remainder one bit-position during the normalizing shift, all being given in time units.

Thus a complete iteration will take $T_{add} + T_a + sT_s$ time units, $\underline{s}$ being the number of one bit-position normalizing shifts made during the iteration. Only the fixed-point portion of the square rooting process was simulated, with the operands in the range (1/4, 1). Since floating-point operands are being considered, there is an additional fixed amount of time associated with determining whether the

exponent is odd or even. This would merely shift the timing distributions without altering their essential character. It was assumed that sensing whether the exponent was odd or even and conditionally shifting the operand fraction one bit-position to the right could be done in the time taken to perform a one bit-position shift, and this time cost was accrued whether the right shift occurred or not. In performing the experiment another assumption was made, namely that in the course of examining the floating-point exponents, even and odd exponents occur with equal frequency. Accordingly, then, of the total sample of fraction parts processed, half were taken in the range (1/4,1) and half in (1/2, 1).

In order that a meaningful distribution be obtained, it was important that sensible or typical values be assigned to the parameters T_{add}, T_a, and T_s. The square rooting process consists of a series of subtractions, logical operations, and one bit-position shifts, and therefore if a proper relation between these parameters is used, the problem will be resolved. As a typical example, the execution times of the relevant operations in the IBM 7090 arithmetic unit were used [6]. The fixed-point addition takes 3 clock times, whether the operands possessed

like or unlike signs. Since we are using the 1's complement representation for negative numbers internal to the process, no additional recomplementation time is required to obtain a signed magnitude form as is done in the IBM 7090. It may be desirable in certain instances, however, to recomplement the final remainder and present it as output information in a register at the conclusion of the square root operation, but this was not done in the experiment. One single bit-position shift in the IBM 7090 arithmetic unit is performed in one clock time, and thus the add-to-shift ratio is obtained. Since in our equipment it was postulated that the logical operations of augmenting the partial root and forming the new trial factor could be accomplished simultaneously in the time required to perform a one bit-position shift, the problem can now be fully specified. Therefore, if $T_{add} = 3$ and $T_a = T_s = 1$ time unit, the parameters (3,1,1) will describe a meaningful problem.

The probability density and cumulative distribution functions for this problem were obtained from a simulation program for the IBM 7090 (see Appendix), and are displayed in Figure 3-8. 2^{14} operands were processed, and with the parameters used no operand took less than 42 time units to

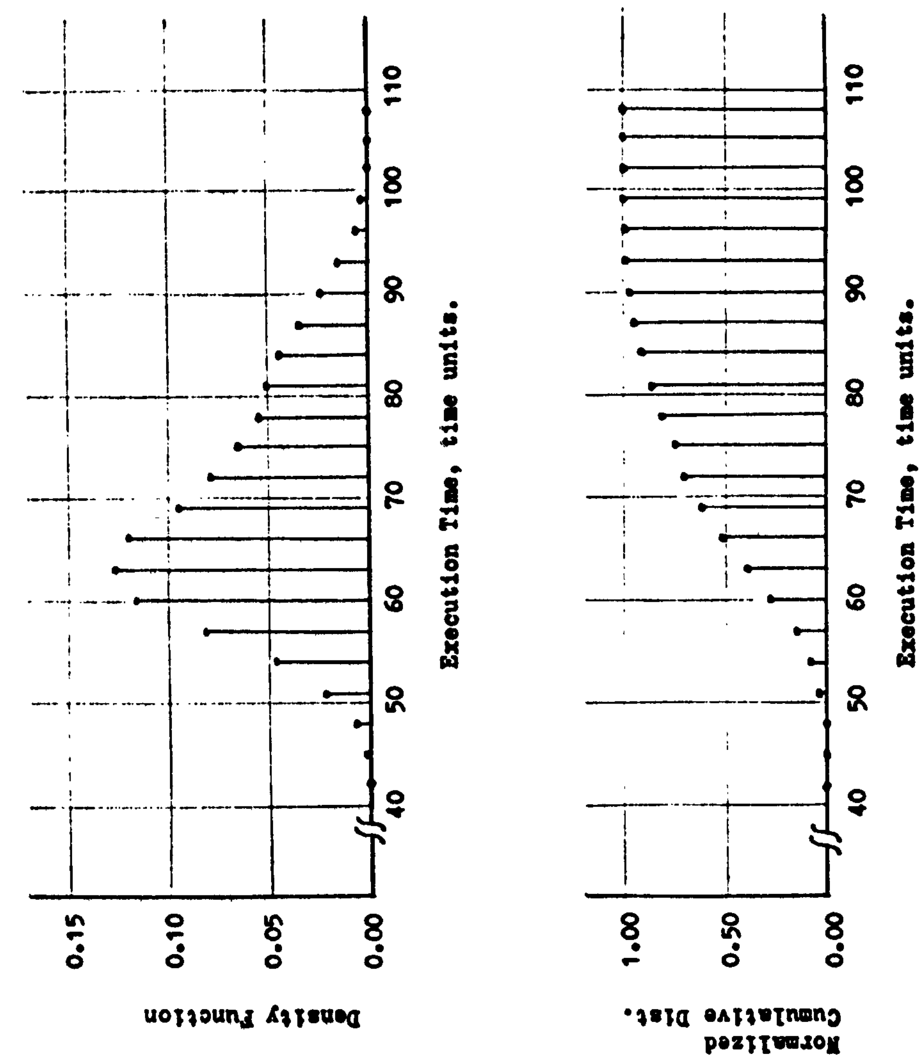


Figure 3-8: Statistical Timing Distributions for the Binary Square Root, Binomial Theorem Method, 1's Complement Negative Numbers, Parameters (3,1,1).

execute, and none more than 108. It is seen that the distribution of execution time is skewed to the right, and for the purposes of graphical analysis, i.e., to determine the mean and variance, it is convenient to make a transformation of variables such that a function $\phi(t)$ of the execution time t becomes normally distributed. Such a transformation is [5]

$$\phi(t) = \frac{g(t) - g(\mu_t'')}{\sigma_t} , \tag{10}$$

where g(t) includes no unknown parameters. The cumulative distribution function for execution time, when plotted as in Figure 3-8, gives the probability that a randomly-chosen operand of the type considered will take more than (or less than) a specified number of time units to have its square root extracted by the binomial theorem method. The cumulative distribution is plotted on a normal probability scale in Figure 3-9, and is plainly skew. If, however, the cumulative distribution of $\log_{10} t$ is plotted as in Figure 3-10, it is found that this distribution may be approximated by a straight line, and thus the variable $(\log_{10} t - \log_{10} \mu_t'')/\sigma_t$ is approximately normally distributed, where μ_t'' is the median of t and σ_t is the standard deviation of $\log_{10} t$. From Figure 3-10, the median is about 66 time

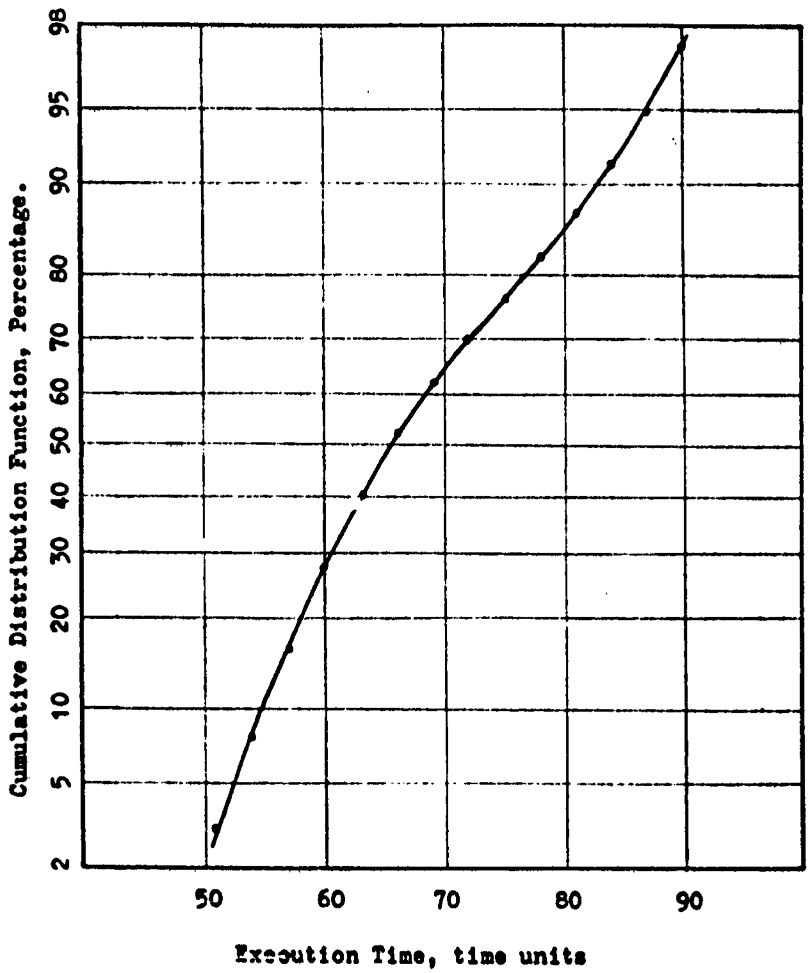


Figure 3-9: Cumulative Distribution Function for Binary Square Root.

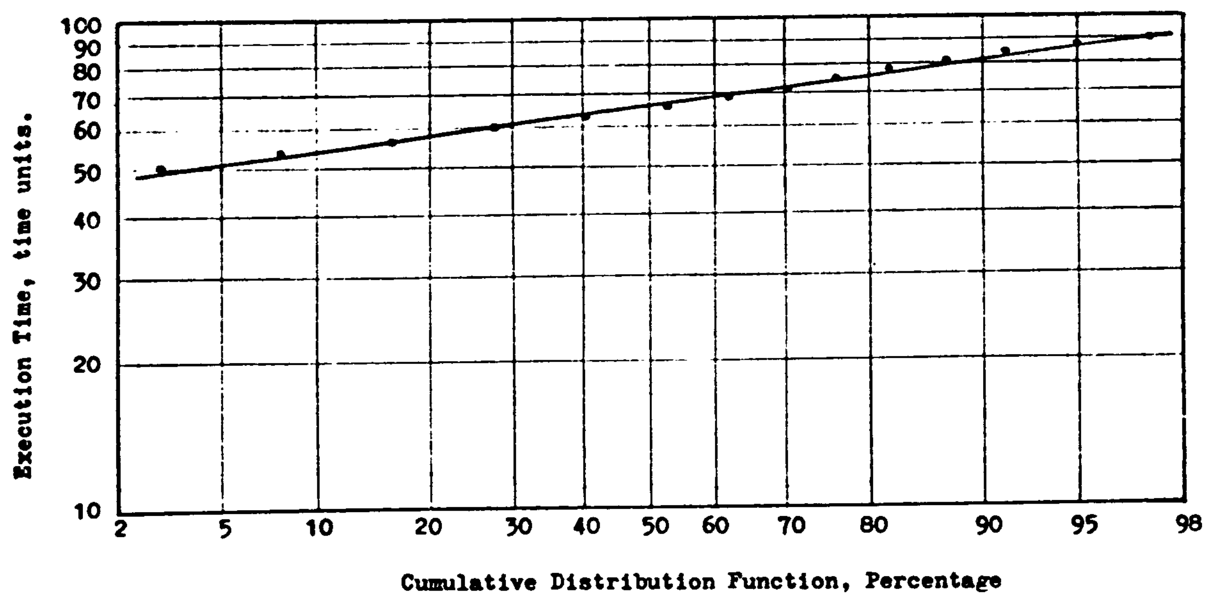


Figure 3-10: Cumulative Distribution Function, Logarithmic Scale.

units. The mean is given by $\mu_t = \mu_t'' \, 10^{\sigma_t^2/2M}$, where $M = \log_{10}e = 0.4343$. To compute the standard deviation of $\log_{10}t$, note the values of t where the cumulative distribution is equal to 0.159 and 0.841; these values are 57 and 77 time units. Taking the average value, $\sigma_t = \frac{1}{2}(\log_{10}77 - \log_{10}57)$, or about 0.065. The mean μ_t is then about 67 time units. The average standard deviation of t is $\frac{1}{2}(77 - 57)$, or about 10 time units. A direct computation using the experimental data yielded a sample mean of 68.8 time units and a standard deviation of 10.6 time units, both values being verified by their graphical estimates.

It then can be concluded that a randomly-chosen floating-point binary operand of the format chosen has an expected execution time of about 69 time units with standard deviation 10.6, when processed by a square rooter of the type described, a time unit being the time necessary to perform a one bit-position shift. The minimum execution time is 42 time units, and the maximum 108, on the order of 3.5 and 9 IBM 7090 machine cycles, respectively. This compares rather favorably with the 67 cycles needed by the SHARE program described in Chapter I.

CHAPTER IV

Other N^{th} Rooting Methods

The binomial theorem method obviously lent itself to direct mechanization of the square root operation. In this chapter the properties of other n^{th} rooting procedures will be considered, to provide a foundation for comparison with respect to mechanization parameters.

4-1: The Euler Iteration Formulae

In a recent article [13] , J. F. Traub has outlined a method for generating iteration formulae of arbitrary order, along with an error estimate. The following development is essentially his as given in his paper.

Let us start by desiring a real root of the function $y = f(x) = 0$ and denote this root as α , so that $f(\alpha) = 0$. The only assumption that is made is that α be a root of multiplicity one. Given the inverse relations

$$y = f(x) \quad , \quad x = g(y) \quad , \tag{1}$$

then
$$g(0) = g(y_i - y_i) \quad . \tag{2}$$

Expanding (2) in a Taylor series gives

$$\alpha = \sum_{k=0}^{\infty} \frac{(-)^k}{k!} y_i^k g^{(k)} \quad , \tag{3}$$

where the parenthized superscript denotes a higher deriva-

tive. Since $g(y_i) = x_i$, (3) reduces to

$$\alpha = x_i + \sum_{k=1}^{\infty} \frac{(-)^k}{k!} y_i^k g^{(k)} \quad . \tag{4}$$

Defining $\quad u = f(x_i)/f'(x_i)$ and $\quad$ (5)

$$Y_k = \frac{(-)^k}{(k+1)!} \{f'(x_i)\}^{k+1} g^{(k+1)} \quad , \tag{6}$$

(4) takes the more compact form

$$\alpha = x_i - u \sum_{k=0}^{\infty} u^k Y_k \quad . \tag{7}$$

If we then take only the first m+1 terms of the series in (7), and denote the right side of (7) as a better approximation to α than x_i (assuming that the sequence of approximations converges), the following iteration formula is a natural consequence:

$$x_{i+1} = x_i - u \sum_{k=0}^{m} u^k Y_k \quad . \tag{8}$$

Defining the Euler polynomial as

$$Y(u) = \sum_{k=0}^{m} u^k Y_k \tag{9}$$

transforms (8) into

$$x_{i+1} = x_i - uY(u) \quad . \tag{10}$$

Defining

$$D_k = \frac{f^{(k)}(x_i)}{f'(x_i)} \quad , \tag{11}$$

Traub shows that Y_k is a polynomial in $D_1, D_2, \ldots, D_{k-1}$, where

$$D_k = D_2 D_{k-1} + \frac{d}{dx} D_{k-1} \quad , \quad k>1 \; ,$$
$$D_1 = 1 \; , \tag{12}$$

such that

$$Y_0 = 1$$
$$Y_1 = (1/2)D_1 \tag{13}$$
$$Y_2 = (1/2)\,D_2^2 - (1/6)D_3 \quad , \text{ etc.}$$

The error of the iteration formula (10) may be estimated by considering the error $\epsilon_{i+1} = \alpha - x_{i+1}$, the remainder of the truncated series in (8):

$$\epsilon_{i+1} = u \sum_{m+1}^{\infty} u^k \, Y_k \quad . \tag{14}$$

If f(x) is a smooth curve in the neighborhood of $x = \alpha$, we may write

$$f(\alpha+\epsilon) \approx f(\alpha) + \epsilon f'(\alpha) \quad ,$$

where ϵ is a small error. On the i^{th} iteration, $x_i = \alpha + \epsilon_i$, and since $f(\alpha) = 0$, $f(x_i) \approx \epsilon_i f'(\alpha)$. Since f(x) is smooth, $f'(x_i) \approx f'(\alpha)$, and thus the error may be estimated as

$$\epsilon_i \approx f(x_i)/f'(x_i) = u \quad . \tag{15}$$

Thus $u^k \approx \epsilon_i^k$, and so

$$\epsilon_{i+1} = \sum_{m+1}^{\infty} Y_k \, \epsilon_i^{k-1} \quad .$$

Expanding Y_k in a power series about α, and assuming that $\epsilon_i^{k+1} \ll \epsilon_i^k$,

$$\epsilon_{i+1} \approx Y_{m+1}(\alpha)\,\epsilon_i^{m+2} \;, \quad m = 0,1,2,\ldots \tag{16}$$

Thus, for a given value of m, an iteration formula of order m+2 may be obtained from (8), with error estimate (16).

In an earlier paper [12] Traub compared various iterative methods for the calculation of n^{th} roots, and introduced an iterative formula which he called "multiterm" iteration, an iteration formula which may be derived from the Euler formula. Multiterm iteration considers the special equation $f(x) = x^n - A$, where $f(\alpha) = 0$, with

$$\alpha = A^{1/n} = x(1 - f/x^n)^{1/n} \quad . \tag{17}$$

Letting $v = -f/xf'$, $\alpha = x(1 + nv)^{1/n}$, or

$$\alpha = x + x \sum_{k=1}^{\infty} \binom{1/n}{k} n^k v^k \quad . \tag{18}$$

Noting that $v = -u/x$,

$$\alpha = x + x \sum_{k=1}^{\infty} (-)^k n^k \binom{1/n}{k} u^k x^{-k} \; . \tag{19}$$

Using f(x) as given above,

$$D_k = f^{(k)}/f' = (n-1)(n-2)\cdots(n-k+1)x^{-k+1} \; . \tag{20}$$

Comparing (19) with (7), using (20) gives

$$Y_k = (n-1)(2n-1)\cdots(kn - 1)x^{-k}/(k+1)!, \quad k = 0,1,2,\ldots \tag{21}$$

for this special case. Multiterm iteration may be made any order by considering only part of the infinite series in

(18). Specifically, the iteration formula of order m is

$$x_{i+1} = x_i + x_i \sum_{1}^{m-1} a_k v^k \quad , \tag{22}$$

where

$$a_k = \left\{\frac{n+1}{k} - n\right\} a_{k-1} \quad , \quad \begin{array}{l} a_0 = 1 \\ k = 1,2,3,\ldots \end{array} \tag{23}$$

The upper bound on the error is

$$\epsilon_{i+1} < \frac{1}{m} \left\{\frac{n}{\alpha}\right\}^{m-1} \epsilon_i^m \quad , \quad m = 2,3,4,\ldots \tag{24}$$

Traub points out that the multiterm iteration formula may be applied in a sequence such that the order of each succeeding application may or may not be changed, until the root has been computed to the desired precision.

Rational Approximations to the Euler Polynomial

In his paper, Traub also considers rational approximations to the Euler polynomial of a form due to Padé. Written this way,

$$Y(u) \approx P(u)/Q(u) \quad , \tag{25}$$

where

$$P(u) = \sum_{0}^{p} u^k P_k \text{ , and} \tag{26}$$

$$Q(u) = \sum_{0}^{q} u^k Q_k \quad . \tag{27}$$

Equation (10) may be written

$$x_{i+1} = x_i - uP(u)/Q(u) \quad . \tag{28}$$

Writing (7) as $\alpha = x_i - uY(u) - E$, (29)

where $E \approx Y_{m+1}\epsilon_i^{m+2}$, and subtracting (28) from (29) gives

$$\alpha - x_{i+1} = \epsilon_{i-1} \approx -u\left\{P(u)/Q(u) - Y(u)\right\} + E$$

or $\epsilon_{i+1} \approx -uH(u)/Q(u) + E$, where

$$H(u) = P(u) - Y(u)Q(u) = \sum_k H_k u^k . \tag{30}$$

Referring to (30), if the leading term of $H(u)/Q(u)$ is proportional to u^{m+1}, then analogous to (16), the iteration formula (28) is of order $m+2$. Thus Traub chooses the $p+q+1$ parameters P_k, Q_k so that $H_k = 0$, $k=0,1,2,\ldots, p+q$, with $p+q=m$. To do this, equate like powers of u in (30), using the series in (26) and (27). Traub gives the resulting equation

$$P_r w_{rp} - \sum_{k=0}^{s} Q_k Y_{r-k} = 0 , \tag{31}$$

where

$$w_{rp} = \begin{cases} 1 & r \leq p \\ 0 & r > p \end{cases} , \tag{32}$$

and

$$s = \min.(r, q) . \tag{33}$$

Thus (31) can be used to find the P_k and Q_k recursively, since the Y_k are known (eqn. (13)), and $P_0 = 1$. Traub then gives the corresponding error formula

$$\epsilon_{i+1} \approx (Y_{m+1} - H_{m+1})\epsilon_i^{m+2} , \tag{34}$$

which indicates an iterative formula of order $m+2$, where

$$H_{m+1} = -\sum^{q} Q_k Y_{m-k+1} . \tag{35}$$

The iterative formula (27) may then be written in the compact form

$$x_{i+1} = I_{pq}(x_i) \quad , \tag{36}$$

where $I_{pq}(x_i)$ is defined as

$$I_{pq}(x_i) = x_i - u\,\frac{P(u)}{Q(u)} \;; \quad \begin{matrix} p = 0,1,2,\dots,m \\ q = 0,1,2,\dots,m \\ p+q = m\,. \end{matrix} \tag{37}$$

Equation (36) then defines m+1 iterative formulae, a few of which are summarized below:

1). $m = 0$:

$$I_{00} = x - u \;; \quad \epsilon_{i+1} = Y_1\,\epsilon_i^2 \tag{38}$$

2). $m = 1$:

$$I_{10} = x - u(1 + Y_1 u) \;; \quad \epsilon_{i+1} = Y_2\,\epsilon_i^3 \tag{39}$$

$$I_{01} = x - \frac{u}{1 - Y_1 u} \;; \quad \epsilon_{i+1} = (Y_2 - Y_1^2)\,\epsilon_i^3 \tag{40}$$

3). $m = 2$:

$$I_{20} = x - u(1 + Y_1 u + Y_2 u^2) \;; \quad \epsilon_{i+1} = Y_3\,\epsilon_i^4 \tag{41}$$

$$I_{11} = x - u\,\frac{Y_1 + u(Y_1^2 - Y_2)}{Y_1 - Y_2 u} \;; \quad \epsilon_{i+1} = \frac{Y_3 Y_1 - Y_2^2}{Y_1}\,\epsilon_i^4 \tag{42}$$

$$I_{02} = x - \frac{u}{1 - Y_1 u + (Y_1^2 - Y_2)u^2} \;;$$

$$\epsilon_{i+1} = (Y_3 - 2Y_1 Y_2 + Y_1^3)\,\epsilon_i^4 \tag{43}$$

In the above formulae, $x = x_i$, and in the error estimates the Y_k are evaluated at the n^{th} root α. The formulae I_{m0}

are those which result from equation (10), the iteration formula before the Pade approximation was applied. For the particular example $f(x) = x^n - A$, Traub indicates that the formulae of the form I_{mm} are preferable from the standpoint of error estimate. A remark by Kogbetliantz [9] also states that rational approximations of this form are the most useful.

<u>Specialization to the Extraction of n^{th} Roots</u>

In order to apply the above methods to the extraction of integral roots, the particular equation $f(x) = x^n - A$ must be considered. The Y_k for any particular n are given in equation (21), and the first few are

$$
\begin{aligned}
Y_0 &= 1 \\
Y_1 &= (n-1)/2x \\
Y_2 &= (2n^2 - 3n + 1)/6x^2 \\
Y_3 &= (6n^3 - 11n^2 + 6n - 1)/24x^3 \quad \text{, etc.}
\end{aligned} \tag{44}
$$

Also,

$$u = \frac{f}{f'} = \frac{x^{-n+1}}{n}(x^n - A) \quad . \tag{45}$$

Using (44) and (45) to write out the first few iteration formulae gives

$$I_{00} = \frac{1}{n}\left\{(n-1)x - \frac{A}{x^{n-1}}\right\} \tag{46}$$

$$\epsilon_{i+1} = \frac{n-1}{2\alpha}\,\epsilon_i^2 \leq (n-1)\,\epsilon_i^2 \tag{47}$$

$$I_{10} = x\left\{1 - \frac{1}{n}\left(1 - \frac{A}{x^n}\right) - \frac{n-1}{2n^2}\left(1 - \frac{A}{x^n}\right)^2\right\} \tag{48}$$

$$\epsilon_{i+1} = \frac{2n^2 - 3n + 1}{6\alpha^2}\,\epsilon_i^3 \leq \tfrac{1}{3}(4n^2 - 6n + 2)\,\epsilon_i^3 \tag{49}$$

$$I_{01} = x\left\{\frac{(n-1)x^n - (n-1)A}{(n-1)x^n - (n-1)A}\right\} \tag{50}$$

$$\epsilon_{i+1} = \frac{n^2 - 1}{12\alpha^2}\,\epsilon_i^3 \leq \tfrac{1}{3}(n^2 - 1)\,\epsilon_i^3 \tag{51}$$

$$I_{20} = I_{10} - x\left\{\frac{2n^2 - 3n + 1}{6n^3}\left(1 - \frac{A}{x^n}\right)^3\right\} \tag{52}$$

$$\epsilon_{i+1} = \frac{6n^3 - 11n^2 + 6n - 1}{24\alpha^3}\,\epsilon_i^4 \leq \tfrac{1}{3}(6n^3 - 11n^2 + 6n - 1)\,\epsilon_i^4 \tag{53}$$

$$I_{11} = x\left\{1 - \frac{1}{n}\left(1 - \frac{A}{x^n}\right)\frac{(7n - 1)x^n - (n - 1)A}{(2n - 2)x^n - (4n - 2)A}\right\} \tag{54}$$

$$\epsilon_{i+1} = \frac{2n^3 - n^2 - 2n + 1}{72\alpha^3}\,\epsilon_i^4 \leq \tfrac{1}{9}(2n^3 - n^2 - 2n + 1)\,\epsilon_i^4 \tag{55}$$

$$I_{02} = x\left\{\frac{(5n^2 + 5n + 1)x^{2n} + (8n^2 - 5n - 2)x^n + (1 - n^2)A^2}{(5n^2 + 6n + 1)x^{2n} + (8n^2 - 6n - 2)x^n + (1 - n^2)A^2}\right\} \tag{56}$$

$$\epsilon_{i+1} = \frac{n^3 + n + 2}{24\alpha^3}\,\epsilon_i^4 \leq \tfrac{1}{3}(n^3 + n + 2)\,\epsilon_i^4 \tag{57}$$

As is expected, the iterative formulae become more compli-

cated as their order increases, and higher order formulae may be derived from an extension of (44) and from (45).

4-2: The Padé Table of Rational Approximations [9]

This method enables a general power series, whether convergent or divergent, to be approximated by a rational function of the form $R_{rs} = P_r(x)/Q_s(x)$, where

$$P_r(x) = \sum_{0}^{r} a_k x^k \quad , \tag{58}$$

$$Q_s(x) = 1 + \sum_{1}^{s} b_k x^k \quad . \tag{59}$$

We desire the approximation

$$f(x) = \sum_{0}^{\infty} c_k x^k \approx R_{rs}(x) = P_r(x)/Q_s(x) \quad , \tag{60}$$

and if the definition

$$Q_s(x) \sum_{0}^{\infty} c_k x^k - P_r(x) = x^{r+s+1} \sum_{0}^{\infty} \gamma_k x^k \tag{61}$$

is imposed, the coefficients a_k and b_k may be found from the resulting linear system of $r+s+1$ equations. In general, the accuracy of the approximation $R_{rs}(x)$ increases as the degree of $P_r(x)$ and $Q_s(x)$ increases. According to E. G. Kogbetliantz [9] , the entries in the r by s table which are the most useful are those for which $r = s$ or $r = s+1$. If $r = s$, then $a_0 = c_0$, and

$$\sum_{r=0}^{s} b_r c_{s-r+1} = 0 \tag{62}$$

$$a_i = \sum_{r=0}^{i} b_r c_{i-r} \quad , \quad i = 1,2,3,\ldots, s \ . \tag{63}$$

and

$$\gamma_k = \sum_{r=0}^{s} b_r c_{2s+k+1-r} \quad , \quad k = 0,1,2,\ldots \tag{64}$$

The γ_k decrease extremely rapidly, and thus

$$x^{2r+1} \sum_{0}^{\infty} \gamma_k x^k \approx \gamma_0 x^{2r+1} \quad .$$

Therefore as a rough estimate $(r = s)$,

$$E_r(x) = \sum_{0}^{\infty} c_k x^k - \frac{P_r(x)}{Q_r(x)} \approx \frac{\gamma_0 x^{2r+1}}{Q_r(x)} \quad . \tag{65}$$

Furthermore, $Q_r(x) \approx 1$, and thus

$$E_r(x) \approx \gamma_0 \ x^{2r+1} \quad . \tag{66}$$

Since the range of x and the order r are presumed to be known, a rough estimate of the error may be obtained by computing γ_0. If $0 \leq x \leq x_0$,

$$|E_r| \leq |\gamma_0| \ x_0^{2r+1} \quad . \tag{67}$$

γ_0 is obtained by solving the system of r+1 equations (62) and (64) with $r = s$, $k = 0$:

$$\sum_{r=0}^{s} b_r c_{s-r+i} = 0 \ , \quad i = 1,2,3,\ldots$$

and

$$\gamma_0 = \sum_{n=0}^{s} b_r c_{2s+1-r} \quad .$$

This yields $\gamma_0 = \delta_r/\Delta_r$, where

$$\Delta_r = \begin{vmatrix} c_1 & c_2 & \cdots & c_{r+1} \\ c_2 & c_3 & \cdots & c_{r+2} \\ \vdots & \vdots & & \vdots \\ c_{r+1} & c_{r+2} & \cdots & c_{2r+1} \end{vmatrix} ; \quad \delta_r = \begin{vmatrix} c_1 & c_2 & \cdots & c_r \\ c_2 & c_3 & \cdots & c_{r+1} \\ \vdots & \vdots & & \vdots \\ c_r & c_{r+1} & \cdots & c_{2r} \end{vmatrix} \tag{68}$$

δ_r being the principal minor of Δ_r. The approximation $R_{rr}(x) = P_r(x)/Q_r(x)$ may be written as a continued fraction

$$\frac{P_r(x)}{Q_r(x)} = A_0 + \sum_{k=1}^{n} \frac{A_k}{x + B_k +} \quad , \tag{69}$$

and the coefficients A_k, B_k may be found by combining and cross-multiplying (69). An examination of (69) shows that parallel computation enables $R_{rr}(x)$ to be formed in r divisions and r+1 additions.

<u>Specialization to n^{th} Root</u>

Kogbetliantz treats this problem by considering the approximation in a general interval (b, c) using the substitution $x = a(1 + z)$, where $b < a < c$. Then $x^{1/n} = a^{1/n}(1 + z)^{1/n}$ is expanded into a binomial series

$\sum_{0}^{\infty} \binom{1/n}{k} a^{1/n} z^k$, $|z| \leq 1$, and a Padé approximation formed. To further restrict the range of z, let $b = a(1 - r_1)$ and $c = a(1 + r_2)$, $0 < r_1 < 1$, $0 < r_2 < 1$, so that $-r_1 \leq z \leq r_2$, which still satisfies $|z| \leq 1$. Let

$$\Upsilon(z) = \sum_{0}^{\infty} \Upsilon_k z^k \quad . \tag{70}$$

As k gets large, the ratio $\Upsilon_{k+1}/\Upsilon_k$ approaches -1, and thus the series may be approximated by an alternating geometric series which has a known sum. Therefore

$$\Upsilon(z) \approx \Upsilon_0/(1 + z) \quad . \tag{71}$$

Then the error formula (65) may be written

$$E_r(z) \approx \frac{\Upsilon_0 \, z^{2r+1}}{(1 + z) \, Q_r(z)} \quad .$$

The relative error, $E_r(z)/x^{1/n}$, is

$$\hat{E}_r(z) \quad \frac{\Upsilon_0 \, z^{2r+1}}{a^{1/n}(1 + z)^{(n+1)/n} \, Q_r(z)} \quad . \tag{72}$$

Letting $E_r(z) = K\phi(z)$ where K = constant, it has been found that the extrema of the relative error lie at $z = -r_1$ and $z = r_2$. Equating the absolute value of the relative error at these values of z gives $|\hat{E}_r(-r_1)| = |\hat{E}_r(r_2)|$. Written out,

$$\frac{r_1^{2r+1}}{(1-r_1)^{(n+1)/n}\,Q_r(-r_1)} = \frac{r_2^{2r+1}}{(1+r_2)^{(n+1)/n}\,Q_r(r_2)} \quad . \tag{73}$$

The ratio c/b gives a second equation involving r_1 and r_2,

$$c/b = (1 + r_2)/(1 - r_1) \quad , \tag{74}$$

where c/b is a known constant since the interval (b,c) has been specified. Solving (73) and (74) yields the desired values r_1 and r_2, so that the maximum relative error and the constant $\underline{a}$ may be computed. The constants a_0, a_1, a_2, ..., which are functions of $\underline{a}$, are then computed, and then a continued fraction representation may be obtained of the form

$$A_0 + \sum_{k=1}^{n} \frac{A_k}{z + B_k +} \quad . \tag{75}$$

Substituting $z = (x - a)/a$ into (75) gives the desired approximation to $x^{1/n}$. In his article Kogbetliantz gives second order ($r=2$) results for the square root, $n=2$:

$$x^{1/2} \approx \frac{5\sqrt{70}}{14} - \frac{50\sqrt{70}/49}{x+47/14 +}\,,\ \frac{4/49}{x + 3/14}$$

$$0.25 \le x < 0.5 \ , \quad |\hat{E}_2| \le 10^{-5} \quad ,$$

$$x^{1/2} \approx \frac{5\sqrt{35}}{7} - \frac{200\sqrt{35}/49}{x+47/7 +}\,,\ \frac{16/49}{x + 3/7}$$

$$0.5 \le x < 1 \ , \quad |\hat{E}_2| \le 10^{-5} \quad .$$

The accuracy of this type of approximation can be improved

either by using higher order rational approximations or by decreasing the size of the interval in which the approximation is valid. From the standpoint of computing time the latter is preferable, although it results in more storage space being required.

The simplest, though not the most accurate rational approximation which is a function of the operand is

$$f(x) \approx R_{11}(z) = \frac{a_0 + a_1 z}{1 + b_1 z} , \quad x = x(z) , \tag{76}$$

which can be computed in one multiplication, one addition, and one division. In order to use this approach to extract integral roots, let us consider the function $f(x) = x^{1/n}$, $n = 2,3,4,\ldots$, where $x = a(1 + z)$, $|z| \leq 1$. As before,

$$x^{1/n} = \sum_{0}^{\infty} c_k z^k , \quad c_k = a^{1/n} \binom{1/n}{k} ,$$

and

$$(1 + b_1 z) \sum_{0}^{\infty} c_k z^k - (a_0 + a_1 z) \quad z^3 \sum_{0}^{\infty} \gamma_k z^k = \gamma(z) . \tag{77}$$

Solving (77),

$$b_1 = \frac{n-1}{2n}, \quad n = 2,3,4,\ldots , \tag{78}$$

$$c_{k+3} + b_1 c_{k+2} = \gamma_k , \ k = 0,1,2,\ldots \tag{79}$$

With $k = 0$, $\gamma_0 = c_3 + b_1 c_2$, or

$$\gamma_0 = a^{1/n} \left\{ \frac{n^2 - 1}{12 n^3} \right\}, \quad n = 2,3,4,\ldots \tag{80}$$

Considering the approximation in the interval (b, c) as before, with $b = a(1 - r_1)$, $c = a(1 + r_2)$, equating the absolute value of the relative error at $z = -r_1$ and $z = r_2$ gives, since $\Upsilon(z) \approx \Upsilon_0/(1 + z)$,

$$\frac{r_1^3}{(1 - r_1)^{(n+1)/n}(1 - b_1 r_1)} = \frac{r_2^3}{(1 + r_2)^{(n+1)/n}(1 + b_1 r_2)} . \tag{81}$$

Solving simultaneously with (74) yields r_1 and r_2. If (81) is written $K(r_1) = G(r_2)$, the maximum relative error of the first order approximation is

$$|\hat{E}_1(z)| \le \frac{\Upsilon_0}{a^{1/n}} K(r_1) = \frac{n^2 - 1}{12n^3} K(r_1) \quad . \tag{82}$$

Solution for the other constants yields

$$\begin{aligned} a_0 &= a^{1/n} \quad , \\ a_1 &= \frac{n+1}{2n} a^{1/n} \quad , \end{aligned} \tag{83}$$

where $\underline{a}$ may be computed once r_1 is known.

<u>Choice of Interval</u>

Since the order of the rational approximation has been fixed, the only way that its precision can be varied is by varying the end points of the interval of approximation (b, c). In general it is true that the precision of the approximation increases if the interval length c - b

decreases. Let us deal with fixed-point binary operands in the range $(2^{-n}, 1)$, and partition this range into $2^p(2^n-1)$ subintervals of equal length so that these subintervals may be easily identified by logical circuitry. A computation was made using the interval $(2^{-2}, 1)$, subdivided into 24 subintervals. It was found that the greatest relative error occurred in the lowest subinterval, for which c/b 9/8. This is not surprising, since in the lowest subinterval $x^{1/n}$ has its greatest curvature, thus causing the greatest inaccuracy. A calculation of the worst relative error in the subinterval $(2^{-n}, 2^{-n} + 2^{-n-p})$ has been made for the square, cube, and fourth roots ($n = 2,3,4$, respectively), for varying numbers of subintervals. The results are summarized in Table 4-2.

Although the operand is partitioned into $2^p(2^n - 1)$ logically identifiable subintervals (listed as "maximum number of intervals" in Table 4-2), it is apparent that all of these need not be distinguished from one another. For example, consider the square root being taken in the range (1/4, 1) using 3 subintervals (1/4, 1/2), (1/2,3/4), and (3/4, 1). The maximum relative error is a monotonically decreasing function of the lowest subinterval's end point ratio c/b, and thus the above 3 subintervals can be

	Square Root			Cube Root			Fourth Root		
c/b	Maximum No. of Intervals	Maximum Relative Error	Minimum No. of Int.	Max. No. of Int.	Maximum Relative Error	Min. No. of Int.	Max. No. Int.	Maximum Relative Error	Min. No. of Int.
16	-	-	-	-	-	-	1	$5.2\cdot10^{-2}$	1
8	-	-	-	1	$2.8\cdot10^{-2}$	1	-	$2.5\cdot10^{-2}$	-
4	1	$1.0\cdot10^{-2}$	1	-	$8.2\cdot10^{-3}$	-	-	$6.5\cdot10^{-3}$	-
2/1	3	$1.3\cdot10^{-3}$	2	7	$1.0\cdot10^{-3}$	3	15	$8.1\cdot10^{-4}$	10
3/2	6	$2.6\cdot10^{-4}$	4	14	$2.1\cdot10^{-4}$	6	30	$1.6\cdot10^{-4}$	20
5/4	12	$4.4\cdot10^{-5}$	8	28	$3.5\cdot10^{-5}$	11	60	$2.7\cdot10^{-5}$	40
9/8	24	$6.5\cdot10^{-6}$	15	56	$5.1\cdot10^{-6}$	21	120	$4.1\cdot10^{-6}$	81
17/16	48	$8.8\cdot10^{-7}$	30	112	$6.9\cdot10^{-7}$	42	240	$5.5\cdot10^{-7}$	160
33/32	96	$1.5\cdot10^{-7}$	59	224	$1.2\cdot10^{-7}$	84	480	$9.4\cdot10^{-8}$	319
65/64	192	$1.5\cdot10^{-8}$	118	448	$1.2\cdot10^{-8}$	166	960	$9.2\cdot10^{-9}$	639

Table 4-2: Relative Error Characteristics For First Order Pade Approximations to the Square, Cube, and Fourth Roots of Binary Integers.

reduced to 2, (1/4, 1/2) and (1/2, 1), without exceeding the maximum relative error in the lowest subinterval (1/4, 1/2). Similar reductions can be made concerning the other entries in Table 4-2, and these appear as "minimum number of intervals" in Table 4-2. For first order Pade approximations three stored constants are required for each interval, whether the ratio of polynomials or continued fraction representation is used.

If the problem in question is the computation of the n^{th} root of a 27-bit binary integer to an absolute precision of 1 part in 2^{27} (fraction part of IBM 7090 floating-point word), then since the n^{th} root lies in the range (1/2, 1), the maximum relative error is 2^{-26} or approximately $1.49 \cdot 10^{-8}$. For the square root this corresponds to the entry 65/64 in Table 4-2. For this relative error, then, the size of the table of stored constants required for each order root may be determined. These table sizes are given in Table 4-3.

n	No. of Stored Constants	Maximum Relative Error
2	354	$1.47 \cdot 10^{-8}$
3	498	$1.16 \cdot 10^{-8}$
4	639	$0.92 \cdot 10^{-8}$
5	774	$0.75 \cdot 10^{-8}$
6	915	$0.61 \cdot 10^{-8}$
7	1050	$0.55 \cdot 10^{-8}$

Table 4-3: Size of Stored Constant Tables for the Square Through Seventh Roots, First Order Padé Approximation.

4-3: Extensions of Nadler's Method

M. Nadler [7, 8] has outlined an iterative method published by Flower in 1771, which was first used to compute high precision logarithms, but which is also useful in computing the reciprocal or the integral roots of a given number. If we are given the number A, we may find its reciprocal by multiplying it by a series of constants such that

$$A \prod_i c_i \rightarrow 1 \quad . \tag{84}$$

Dividing (84) by A yields the equation that is necessary to compute the reciprocal of A,

$$\prod c_i \rightarrow A^{-1} \quad . \tag{85}$$

Thus (84) and (85), computed separately, form a pair of iterative equations that yield the reciprocal of a given number. These equations may be used to find the quotient B/A by using the pair of equations

$$A \prod c_i \to 1$$
$$B \prod c_i \to BA^{-1} \quad . \tag{86}$$

A modification of this algorithm has been used for division in the Harvard Mark IV computer, and is given by Richards [10] as

$$\frac{N_{i+1}}{D_{i+1}} = \frac{(2 - D_i)N_i}{(2 - D_i)D_i} \quad , \tag{87}$$

where N_0 is the dividend and D_0 the divisor. The iterative method in (87) will converge if $0 < D_0 < 1$, thus making $D_i < D_{i-1} < 1$, $i = 0,1,2,\ldots$

The iterative method described in (84) and (85) may be extended to the computation of n^{th} roots by employing the following extension, developed by Nadler [8] to extract the square root of a number. Let the following product be formed in a given register:

$$A \prod c_i^n \to 1 \quad . \tag{88}$$

Raising (88) to the power $(n-1)/n$ gives

$$A^{(n-1)/n} \prod c_i^{n-1} \to 1 \quad . \tag{89}$$

Multiplying (89) by $A^{1/n}$ then gives

$$A \prod c_i^{n-1} \rightarrow A^{1/n} \quad , \tag{90}$$

and thus the pair of equations (88) and (90), computed separately, form an iterative algorithm which may be employed to extract the n^{th} root of a given number.

<u>Computational Considerations</u>

Nadler points out that the constants c_i may be of the convenient (in the binary number system) form 1 ± 2^{-p}, $p = 1,2,3,\ldots$, so that multiplication may be carried out using a shift and an addition. Richards discusses the Harvard Mark IV division algorithm in the decimal system where the same sort of approximation is used, i.e., $2 - D_i \approx 1 + d_i$, where d_i is the highest order nonzero digit of $1 - D_i$. Suppose that $A \prod c_i \rightarrow 1$ monotonically from below, and thus c_i is of the form $1 + 2^{-p}$. After a few iterations the process will reach a point where $A \prod c_i$ will be of the form $0.1111\cdots$, such that each succeeding iteration will merely add another "1" to the string already obtained. Thus if k significant digits of the quotient are desired, nearly that many shift-addition operations will be required.

Let us examine the precision of these iterative

methods:

1). <u>Division</u>

$$A \prod c_i \to 1$$

$$\prod c_i \to A^{-1} = Q$$

Let

$$A \prod c_i = 1 - \Delta \quad ,$$

then

$$\prod c_i = Q(1 - \Delta) \; , \tag{91}$$

and therefore the relative error of the reciprocal (or quotient) is the same as that of the operation which causes the reciprocal to be formed.

2). <u>n^{th} Roots</u>

$$A \prod c_i^n \to 1$$

$$A \prod c_i^{n-1} \to A^{1/n} = \alpha$$

Let

$$A \prod c_i^n = 1 - \Delta \quad .$$

Raise to the power (n-1)/n,

$$A^{(n-1)/n} \prod c_i^{n-1} = (1 - \Delta)^{(n-1)/n} \quad .$$

Since $\Delta \ll 1$,

$$A^{(n-1)/n} \prod c_i^{n-1} \approx 1 - \frac{n-1}{n} \Delta \quad .$$

Multiply by $A^{1/n} = \alpha$,

$$A \prod c_i^{n-1} \approx \alpha \left\{ 1 - \frac{n-1}{n} \Delta \right\} \quad , \tag{92}$$

and thus the relative error of the n^{th} root is less than the relative error of the forcing expression. Therefore if the desired precision of the n^{th} root is specified, the precision to which the forcing expression must be carried out can be determined.

In the case of the n^{th} rooting algorithms given in equations (88) and (90), the form $c_i^n = 1 + 2^{-p}$ poses some problems. The relation between c_i^n and c_i^{n-1} must be exact or to within the maximum tolerance of the rooting procedure in order that the n^{th} root thus extracted be correct to the specified precision. Richards states that it is desirable to make the capacity of the registers holding the factors in question one or two digits greater than the word length of the reciprocal (or root) in order to minimize the effect of round-off errors. In the case of the square root ($n=2$), the problem may be handled in the following manner:

Let a partial result be given as $A \prod_i^{m-1} c_i^2$, and let this result be used to determine the next multiplying constant $c_m^2 = 1 + 2^{-p}$, $p \geq 1$. Now if p is large enough,

$$c_m = (1 + 2^{-p})^{1/2} \approx 1 + 2^{-p-1} ,$$

thus giving $c_m^2 = 1 + 2^{-p} + 2^{-2p-2}$. Therefore the factor

$A\prod c_i^2$ could be used to determine the squares of the multiplying constants, and thus the constants themselves, both in an exact manner. There is one complication that might arise in the application of the above method, however, namely that $A\prod c_i^2 > 1$. This may be remedied by taking $c_m^2 = 1 \pm 2^{-p} + 2^{-2p-2}$, $c_m = 1 \pm 2^{-p}$, using $c_m = 1 - 2^{-p}$ when $A\prod c_i^2 > 1$ and $c_m = 1 + 2^{-p}$ when $A\prod c_i^2 < 1$. When $A\prod c_i^2 = 1$, the process terminates because an exact root to within the process tolerance has been found. The constants c_m^2 and c_m imply shift-addition operations, and may be utilized in the same manner as in the division process. If k significant digits are to be computed in the square root and δ additional digits are carried along in the computation to counter round-off error, then the effect of 2^{-2p-2} vanishes when $2p+2 > k+\delta$, or $p > \frac{1}{2}(k + \delta - 2)$, approximately the midpoint of the iterative process, and the simpler approximation $c_i^2 = 1 \pm 2^{-p}$ may be used thereafter.

For the cube root ($n = 3$), the approximation to the cube of the constant may be written $c_i^3 = (1 \pm 2^{-p-1})^3 = 1 \pm (2^{-p} + 2^{-p-1}) + (2^{-2p-1} + 2^{-2p-2}) + 2^{-3p-3}$, but this approach is rather impractical, since the approximation c_i must be obtained from $A\prod c_i^3 \rightarrow 1$, and then an exact correspondence between c_i^3 and c_i^2 must be established in order

that the iterative process be valid. It is easily seen that for $n = 4,5,6,\ldots$ this type of approximation defies simple mechanization, since an exact correspondence must be established between c_i^n and c_i^{n-1} after first obtaining an approximation of the form $c_i = 1 \pm 2^{-p-1}$ from the factor $A \prod c_i^n \to 1$.

<u>Stored</u> <u>Tables</u> <u>of</u> <u>Constants</u>

Instead of forming the constants c_i at each stage of the iterative procedure, we could examine the magnitude of $A \prod c_i^n$, and upon the results of this examination, select the appropriate constants c_i^n and c_i^{n-1} from stored tables. The determination of the magnitude of $A \prod c_i^n$ could be made by direct logical access to its bit positions, and thus the appropriate table entries could be selected according to the bit configuration sensed. If k bits of accuracy are desired in the n^{th} root, i.e., $A^{1/n} \leq \alpha(1-2^{-k})$, then according to (92),

$$A \prod c_i^n \approx 1 - \frac{n-1}{n} \cdot 2^{-k} \quad . \tag{93}$$

For example, let us consider extracting the n^{th} root of a k-bit binary integer in the range $(2^{-n}, 1)$ with absolute error less than or equal to 1 part in 2^k. If it is desired to force $A \prod c_i^n$ into the desired range, i.e.,

$$1 - \frac{n-1}{n} \cdot 2^{-k} \leq A \prod c_i^n \leq 1 + \frac{n-1}{n} \cdot 2^{-k} , \tag{94}$$

using just one multiplication, then $2^{k-1} + 2^{k-2} + \cdots + 2^{k-n}$ entries each are required in the c_i^n and c_i^{n-1} tables, making a total of $2^{k+2} - 2^{k-n+1}$ stored constants required. However, since $A \prod c_i^n$ will be in the desired range after one multiplication, the c_i^n table does not have to be stored in this special case since the desired root $\alpha \approx Ac^{n-1}$ may be obtained directly from the c^{n-1} table. If this is the case, about 235 million stored constants would be required to extract the square root of a 27-bit binary integer (such as the fraction part of an IBM floating-point word) in one multiplication, about 252 million to extract the cube root, and even more for the higher roots. These figures are of course entirely out of the question. The number of stored constants required to force $A \prod c_i^n$ into the desired range may be reduced by expending more multiplications, but the c_i^n will have to be stored, and it will require the expenditure of many multiplications in order to reduce the stored tables to a reasonable size.

4-4: Truncated Series Method

Suppose it is desired to compute the value of a function that has a convergent power series representation

$f(x) = b_0 + b_1 x + b_2 x^2 + \cdots$, and suppose further that it is possible to make a transformation on $f(x)$ so that it may be approximated by a severely truncated series, say, $f(x) = b_0 + b_1 x$. It is this type of transformation which will be considered in the computation of the real n^{th} root of a real number.

The binomial expansion

$$(1 + \Delta)^{1/n} = 1 + \frac{1}{n}\Delta + \frac{1}{2!}\,\frac{1}{n}\left(\frac{1}{n} - 1\right)\Delta^2 + \cdots \tag{95}$$

is an alternating power series convergent for $|\Delta| < 1$. Let us suppose that $|\Delta| \ll 1$ so that

$$(1 + \Delta)^{1/n} \approx 1 + \frac{\Delta}{n} \quad , \tag{96}$$

the error being less than the next term, i.e.,

$$|\epsilon| < \frac{n-1}{2n^2}\,\Delta^2 \ , \qquad n = 2,3,4,\ldots \tag{97}$$

Let it be stipulated that our operands are binary integers and that we wish to compute their n^{th} root to an accuracy of at least 1 part in 2^k, i.e., $|\epsilon| < 2^{-k}$. Thus

$$2^{-k} \leq \frac{n-1}{2n^2}\,\Delta^2 \quad ,$$

or

$$\Delta \leq \left\{\frac{2n^2}{n-1}\right\}^{1/2} \cdot 2^{-k/2} \quad . \tag{98}$$

For example, if our operands are IBM 7090 floating-point words with 27-bit fractional parts, then $k = 27$ and the maximum Δ is given in the table below.

n	Maximum Δ	n	Maximum Δ	n	Maximum Δ
2	$1.00 \cdot 2^{-12}$	6	$1.34 \cdot 2^{-12}$	10	$1.66 \cdot 2^{-12}$
3	$1.06 \cdot 2^{-12}$	7	$1.43 \cdot 2^{-12}$	11	$1.74 \cdot 2^{-12}$
4	$1.15 \cdot 2^{-12}$	8	$1.51 \cdot 2^{-12}$	12	$1.81 \cdot 2^{-12}$
5	$1.25 \cdot 2^{-12}$	9	$1.59 \cdot 2^{-12}$	13	$1.87 \cdot 2^{-12}$

Table 4-4: Maximum Value of Δ in the Truncated Series, $k = 27$.

For the values of n shown, $\Delta = 2^{-12}$ is a satisfactory value to use. If we then force our operand into the range $(1, 1 \pm 2^{-12})$, the series given in (96) may be used to compute the n^{th} root of x to within the maximum allowable error.

Transformation of the Operand

Considering that we are operating upon the 27-bit fractional part of IBM 7090 floating-point words, it is given that the operand will be in the range $2^{-n} \leq x < 1$, $n = 2,3,4,\ldots$ It is required to execute some sort of

transformation upon the operand x in order to force it into the interval $(1, 1 \pm 2^{-12})$.

Let us consider a transformation used by Bemer [1] and by Cantor, Estrin, and Turn [2] in the computation of the logarithm of a real number. Let

$$z = x \prod_{1}^{m} c_i \tag{99}$$

define a transformation upon x. Then

$$\ln z = \ln \; x \prod_{1}^{m} c_i = \ln x + \sum_{1}^{m} \ln c_i \;,$$

and thus

$$\ln x = \ln z - \sum_{1}^{m} \ln c_i \quad . \tag{100}$$

The series expansion for ln z about the point z = 1 is

$$\ln z = (z-1) - \tfrac{1}{2}(z-1)^2 + \tfrac{1}{3}(z-1)^3 - \cdots \;, \tag{101}$$

convergent for $0 < z \leq 2$. If $z = 1 + \Delta$, where $\Delta \ll 1$, then

$$\ln(1 + \Delta) \approx \Delta + O(\Delta^2) \;, \tag{102}$$

with error

$$|\epsilon| \leq \tfrac{1}{2} \Delta^2 \quad . \tag{103}$$

Thus if $|z-1| \leq |\Delta|$ by applying the transformation given in (99), ln x may be computed using (100), which employs the severely truncated series in (102). The additional requirement is, of course, that a suitable table of con-

stants $\ln c_i$ be available, as well as the means for extracting the correct entries from this stored table. Cantor, Estrin, and Turn specified an error bound of 2^{-27}, and thus $\Delta \leq 2^{-13}$. They operated upon a normalized ($1/2 \leq x < 1$) 27-bit binary operand with the transformation (99) using two multiplications and two stored tables of constants to force the operand into the range $1 - 2^{-13} < z < 1 + 2^{-13}$. The transformation was defined as $z = a_k c_j x$, where

$$a_k = 2^{-6}\ \text{Int.}\left\{\frac{2^{13}}{k-1}\right\},$$
$$k = \text{Int.}(2^7 x)$$

and

$$c_j = 2^{-13}\ \text{Int.}\left\{2^{26}\ \frac{(1 - 2^{-13})}{j-1}\right\}$$
$$j = \text{Int.}(2^{13} a_k x) \quad .$$

Int.( ) denotes the integer part of the quantity in brackets. Therefore $2^6 \leq k < 2^7$, i.e., $k = 64, 65, \ldots, 127$, and $2^{13} - 2^7 - 2^6 \leq j < 2^{13}$, i.e., $j = 8000, \ldots, 8191$. Thus there are 64 constants a_k and 192 constants c_j required to transform $1/2 \leq x < 1$ into $1 - 2^{-13} < z < 1 + 2^{-13}$, where $z = a_k c_j x$.

In a similar manner, then, let us define a trans-

formation that will force $2^{-n} \leq x < 1$ into $1 - \Delta < z < 1 + \Delta$, where $\Delta = 2^{-12}$ and $z = x \prod c_i$. Let

$$z = x \prod_{1}^{m} c_i \quad , \quad 2^{-n} \leq x < 1 \quad , \tag{104}$$

then

$$z^{1/n} = x^{1/n} \prod_{1}^{m} c_i^{1/n} \quad .$$

Therefore

$$x^{1/n} = z^{1/n} \prod_{1}^{m} c_i^{-1/n} \quad , \tag{105}$$

where $1 - 2^{-12} < z < 1 + 2^{-12}$, and thus $z^{1/n}$ may be computed using the series in (96), with $|\epsilon| \lesssim 2^{-27}$. Consider effecting the transformation (104) in a single multiplication, $z = xa_k$. In order to bring z into the desired range, the first 13 bits of x must be examined. Let $k = \mathrm{Int}(2^{13}x)$ where $2^{-n} \leq x < 1$, and thus $2^{13-n} \leq k < 2^{13}$, $n = 2,3,4,\ldots.$ For each of the a_k we need an $a_k^{-1/n}$ to correct $z^{1/n}$, thus necessitating two tables, a_k and $a_k^{-1/n}$. Table 4-5 gives the total number of stored constants required in the single multiplication scheme.

n	Total no. of const.	n	Total no. of const.	n	Total no. of const.
2	12288	4	15360	6	16128
3	14336	5	15872	7	16256

Table 4-5: Number of Stored Constants Required for nth Root, Single Multiplication Scheme.

Note that the constants a_k have a small number of nonzero bits, and thus if a_k is considered as the multiplier, the computation of $z = xa_k$ is a "short" multiplication. If $n > 13$, either more leading bits of x will have to be examined, necessitating expansion of the stored tables, or an additional multiplication will have to be executed, also introducing additional constants. The present discussion will be limited to the cases where n is not large enough to require such changes.

In order to reduce the number of stored constants required, let us consider forcing z into the desired range using two multiplications, i.e., $z = xa_k c_j$. Following Cantor, Estrin, and Turn, let the transformation sequence be $(2^{-n}, 1) \rightarrow (1 - 2^{-5}, 1 + 2^{-5}) \rightarrow (1 - 2^{-12}, 1 + 2^{-12})$, the respective ranges of x, xa_k, $xa_k c_j$. Define

$$a_k = 2^{-5} \text{ Int.}\left\{ \frac{2^{12}}{k-1} \right\} , \tag{106}$$

$$k = \text{Int.}(2^6 x) \quad , \tag{107}$$

and

$$c_j = 2^{-12} \text{ Int.}\left\{ 2^{24} \ \frac{(1 - 2^{-12})}{j-1} \right\} , \tag{108}$$

$$j = \text{Int.}(2^{12} x a_k) \quad . \tag{109}$$

The ranges of k and j are $2^{6-n} \leq k < 2^6$, $n = 2,3,4,5$, and

$2^{12} - 2^6 - 2^5 \leq j < 2^{12}$. Thus there are no more than 62 constants a_k for $n < 6$, and 96 constants c_j. In addition to these constants, there must be tables of $a_k^{-1/n}$ and $c_j^{-1/n}$ stored. Table 4-6 gives the total number of constants required in the two multiplication scheme for values of n between 2 and 5. If $n > 5$ the a_k and $a_k^{-1/n}$

n	Total no. of const.	n	Total no. of const.
2	288	4	312
3	304	5	316

Table 4-6: Total Number of Constants Required for nth Root, Two Multiplication Scheme.

tables will have to be expanded, with a resultant reduction in the size of the c_j and $c_j^{-1/n}$ tables. The multiplications xa_k and xa_kc_j are "short" and $za_k^{-1/n}$ and $za_k^{-1/n}c_j^{-1/n}$ are regular length.

It should be noted that this "sequential table lookup", abbreviated STL, method as Cantor, Estrin, and Turn call it, is quite similar to Nadler's method for computing roots, in that they both force the operand into a predetermined range. However, the difference between the two methods is the width of this range. In Nadler's method the operand has to be forced into such a narrow range that

either too large a table of stored constants or an unsatisfactory number of multiplications is required.

4-5: Logarithm-Antilogarithm Approach to n^{th} Rooting

If it is required to extract the n^{th} root of a given real number, the following sequence of operations may be performed:

$$\boxed{\ln x} \longrightarrow \boxed{\frac{1}{n} \ln x} \longrightarrow \boxed{x^{1/n} = \exp.\left\{ \frac{1}{n} \ln x \right\}}$$

Figure 4-1: Computational Sequence for the Log-Antilog Method.

The operation e^x is, of course, the antilog operation corresponding to ln x.

Let us examine a variable structure computer developed by Cantor, Estrin, and Turn [2] that computes the elementary functions ln x and e^x. The essential character of their sequential table lookup (STL) algorithm has been given in the section (4-4) dealing with the truncated series method for computing n^{th} roots. Cantor, Estrin, and Turn developed a combined structure that handles both ln x and e^x as well as separate structures, and it is this combined structure whose characteristics will be given.

The constants necessary to compute ln x and e^x

are stored in a table of 1037 words of minimum length 31 bits and maximum length 44 bits. In addition, a 36-bit accumulator, a 35-bit adder, a 36-bit multiplicand register, and a 14-bit MQ register are computational registers required. Besides the necessary memory access hardware required to select the desired constants from memory, there is also the usual control and decoding circuitry that is necessary to make the process function.

CHAPTER V

Comparison of the n^{th} Rooting Methods

5-1: Timing Measures

Each n^{th} rooting method considered is made up of a number of elementary arithmetic and logical operations. However, each method does not necessarily consist of the same operations, and the operations occur in varying proportions according to the method. Therefore, as a first step, the timing evaluations will be made in terms of the elementary operations. The operations used are defined as fixed-point binary, with a fixed word length. Let the following symbols be introduced:

S = one bit-position shift;

A = addition or subtraction;

M = full word-length multiplication;

D = division;

MA = memory access;

M_s = short multiplication, where a short multiplication is one whose multiplier is substantially shorter than the full word length.

5-2: Dealing with the Floating-Point Exponent

It was previously pointed out that the fractional part of a floating-point operand may be shifted as many as n-1 bit positions to the right before execution of a fixed point rooting process, depending upon how nearly

the exponent was a multiple of n. For a general value of n, the only way to determine this property is to perform the division b/n, where b is the exponent, examine the remainder r ($b/n = \text{Int.}\left\{b/n\right\} + r/n$), and shift the fraction part n-r places to the right if r is nonzero. The root exponent is $\text{Int.}\left\{b/n\right\} + 1$ if $r > 0$ and b/n if $r = 0$. For an IBM floating point word, the division b/n is a maximum of 8 bits long, and thus the maximum time taken to deal with the exponent is this 8-bit division plus n-1 one bit position shifts. Therefore, this time must be added onto the maximum expected execution times of those methods which employ operations on just the fractional parts of a floating-point word. These methods are the binomial theorem method, the Euler iteration formulae, the truncated series method, and the Padé approximation.

5-3: The Binomial Theorem Method

A sub-unit of the binomial theorem n^{th} rooting process, an iteration, has been previously defined as:

1). formation of the trial factor;
2). formation of the correction if the remainder is negative;
3). addition/subtraction of the trial factor and correction to the remainder;
4). shifting out leading zeros from the new remainder; and

5). augmenting the partial root with the appropriate bits according to the results of steps 3 and 4.

An iteration is represented schematically in Figure 5-1. The most time-consuming part of the iteration occurs in forming the trial factor and the correction at the beginning of the iteration. For the n^{th} root, the trial factor is a polynomial of degree n-1 in the partial root a_{j-1}, and the correction is a polynomial of degree n-2 in a_{j-1}, the coefficients being the binomial coefficients multiplied by a power of 2 in the case of the trial factor and integers of approximately the same magnitude as the binomial coefficients multiplied by a power of 2 in the case of the correction. Since the trial factor is a higher degree polynomial than the correction, the formation of the trial factor is the longer operation of the two. What is required, then, is to form successively the powers of a_{j-1}, from the square to the $(n-1)^{st}$, and form the trial factor and correction polynomials using the appropriate coefficients.

A highly parallel method of doing this is shown in figure 5-2. The trial factor is represented symbolically as $c_0 + c_1 a_{j-1} + \cdots + c_{n-1} a_{j-1}^{n-1}$, and the correction as $c_0' + c_1' a_{j-1} + \cdots + c_{n-2}' a_{j-1}^{n-2}$, where $c_0, c_1, \cdots, c_{n-1}$; $c_0', c_1', \cdots, c_{n-2}'$ are short integers times a power of 2. Assuming the positionings can be accomplished in one or a

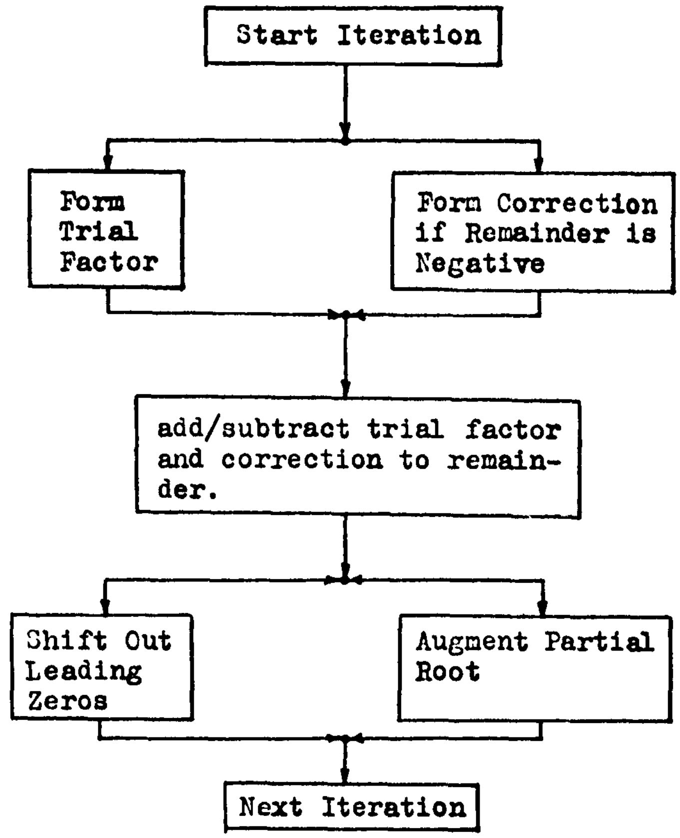


Figure 5-1: Schematic Representation of an Iteration.

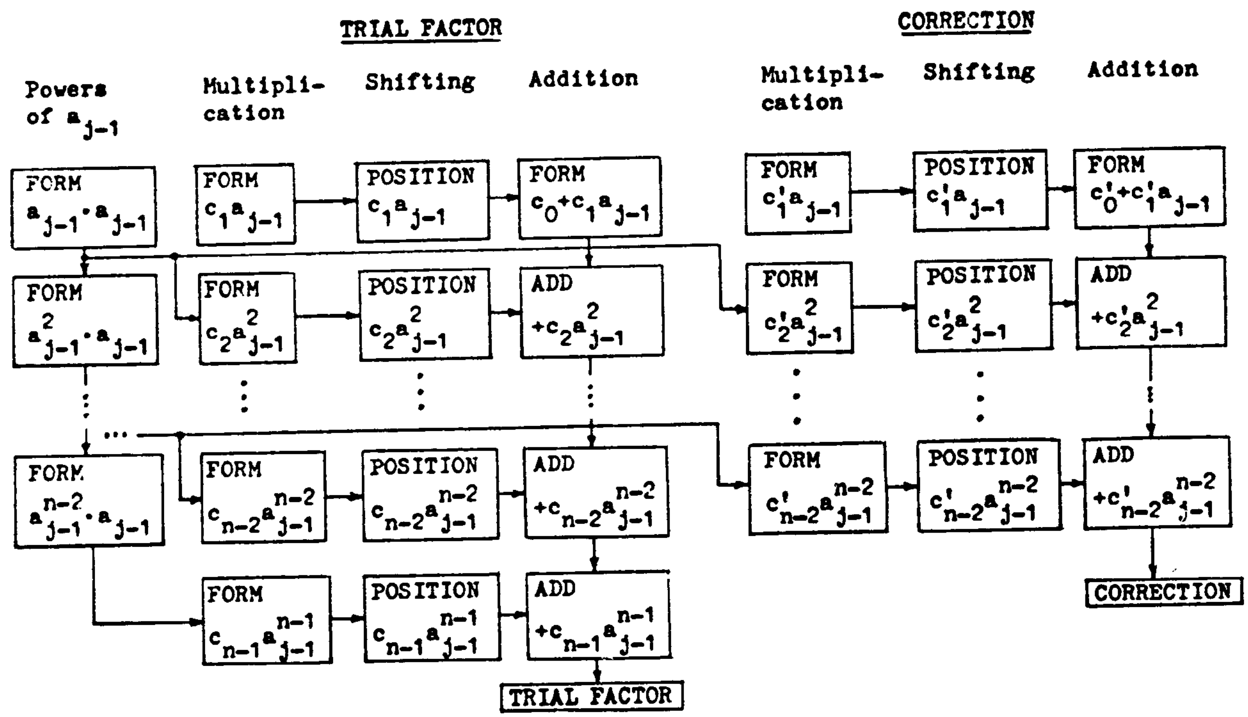


Figure 5-2: Formation of the Trial Factor and the Correction in a Highly Parallel Fashion.

few one bit-position shift times, the entire process of forming the trial factor and correction can be done in the time it takes to form the n-2 powers of a_{j-1}, plus the time taken to form the last term of the trial factor. Done in this way, the arithmetic units which might be used for the formation of the trial factor and the correction are 3 multipliers, 2 multiple place shifting matrices, and 2 adders. During the early stages of the rooting process the partial root a_{j-1} consists of only a few digits, and near the end consists of nearly the full word length. Thus, the n-2 multiplications used to form the powers of a_{j-1} have multipliers with an expected length of one-half the full word length, and therefore are, on the average, short multiplications.

If more conservatively, a single arithmetic unit is used, assuming also that one shifting matrix is available to execute the various variable length shifts required, the computation of the trial factor and correction polynomials takes 3n-5 short multiplications, 2n-3 additions, and 2n-3 variable length shifts, for n = 3, 4, 5, ... To obtain a maximum time estimate, the minimum figure of merit of 1.00 root bits per iteration could be assumed, and thus n^{th} rooting process could take as many as k iterations (k being the number of bits in the fraction part of the floating-point word), each

iteration consisting of $3n-5$ short multiplications, $2n-3$ variable length shifts, 1 bit-position shift time to augment the partial root, and $2n-2$ additions. To extract the n^{th} root of a floating-point binary operand, then it will take a maximum of $k\{(3n-5)M_s + (2n-3)S^* + S + (2n-2)A\} + (n-1)S + D(8)$, where S^* is a variable length shift executed by a shifting matrix and $D(8)$ is an 8-bit division, for $n = 3, 4, 5, \ldots$. If a shifting matrix is not employed, the rooting process for $n > 2$ becomes extremely time consuming due to the large number of sequential one bit-position shifts needed to position the terms of the trial factor and correction. The square root ($n=2$) has been treated as a special case in Chapter III.

5-4: The Euler Iteration Formulae

The computational speeds of the Euler iteration formulae depend upon their order (and thus complexity), and upon the number of times they must be applied. Since the number of applications (or iterations) depends upon the precision desired and the order of the root desired, timing evaluations will be made on a "per iteration" basis and iterations may be cascaded to meet the computational needs of particular problems.

The first six Euler iteration formulae, i.e., those described earlier, will be considered. Table 5-1 gives the execution time of one iteration, $x_{i+1} = I_{pq}$,

using sequential computation with a single arithmetic unit. All operations are fixed-point binary, and "red tape" and data transfer operations are neglected. Also, the time taken to deal with the floating-point exponent is not included in the timing table. Table 5-1 was compiled for a fixed n, i.e., all the expressions containing n were precomputed and assumed available at the time they were required.

Approx.	A	M	M_s	D	P*
I_{00}	1	n-1	0	1	n-2
I_{10}	3	n+3	0	1	n-1
I_{01}	2	2n-1	4	1	2n-2
I_{20}	4	n+6	0	1	n-1
I_{11}	4	n+1	4	2	n-1
I_{02}	4	2n+3	6	1	2n-2

Table 5-1:

Computational Properties of the First Six Euler Formulae, Sequential Computation Using One Arithmetic Unit, n Fixed.

*P = No. of mult. used to form powers of x.

If n becomes substantially large, the computation of x^n takes the major portion of the iteration computation time. Therefore, there is a point at which the computation of a single Euler iteration becomes more time consuming than using another method to compute the n^{th} root, and thus the computation of x^n enters as a limiting factor in the usefulness of the Euler iteration formulae.

5-5: <u>The Padé Approximation Method</u>

The first-order rational approximations considered could take two equivalent forms, either

$$1). \qquad x^{1/n} = \frac{a_0 + a_1 x}{1 + b_1 x}\text{, or}$$

$$x^{1/n} = A_0 + \frac{A_1}{x + B_1} \quad .$$

However, even though the two representations yield equal results to the desired precision, they are not computational equals. Sequential computation of the first representation (ratio of polynomials) takes 2M + 2A + 1D + 3MA + (n-1)S + D(8), and the second (continued fraction) 2A + 1D + 3MA + (n-1)S + D(8). Clearly the continued fraction representation is preferable timewise, the execution times given being those for a floating-point operand.

5-6: Rejection of Nadler's Method

Although they are theoretically sound, the higher order extensions of Nadler's method for calculating n^{th} roots present unreasonable demands in storage (such as several million stored constants being required in a sequential table lookup scheme), or are grossly inconvenient or impossible to mechanize as in the case of the bit-by-bit method of forcing the factor $A \prod c_i^n$ to unity, because of the exact relationship demanded between c_i^n and c_i^{n-1} .

The similarity between Nadler's method and the truncated series method points up the superiority of the latter as far as the number of stored constants required, since in the truncated series method the quantity being forced to unity does not have to approach this value as closely as in Nadler's method, and although more arithmetic operations are expended, the number of stored constants required for the sequential table lookup approach in the truncated series method is far less.

Therefore, Nadler's method is regarded as grossly undesirable in view of the much simpler and more efficient n^{th} rooting methods available, and will be eliminated from further consideration.

5-7: <u>The Truncated Series Method</u>

By applying the transformation $z = x\prod_i c_i$ to the operand x in order to force z into the range $(1 - |\Delta|, 1 + |\Delta|)$, it was shown that $z^{1/n}$ could be computed using the severely truncated series $z^{1/n} \approx 1 + \Delta/n$, where $|\Delta|$ was chosen to satisfy an error criterion. The transformation was accomplished in essentially the number of short multiplications necessary to force z into the desired range, and an equal number of "correcting" full word-length multiplications were applied to $z^{1/n}$ in order to obtain $x^{1/n}$.

The computational sequence is given in Figure 5-2. The two previously discussed transformations were the

single- and two-multiplication types, applied in the case where $|\Delta| \leq 2^{-12}$ to satisfy $|\epsilon| \leq 2^{-27}$. Since $z^{1/n}$ has 28 significant bits in the case of an IBM floating-point binary word, 27 of them to the right of the binary point, and since $|\Delta| \leq 2^{-12}$, the division Δ/n need only be carried out 14 places at the most, depending upon the value of n.

$$\boxed{z = x \prod c_i} \rightarrow \boxed{z^{1/n} = 1 + \frac{\Delta}{n}} \rightarrow \boxed{x^{1/n} = z^{1/n} \prod c_i^{-1/n}}$$

Figure 5-2: Computational Sequence of the Truncated Series Method, Mantissa Part.

Thus Δ/n is a "short" division, and for the sake of argument will be considered as one-half a full word-length division. Another point arises, namely, whether Δ is positive or negative. If $\Delta > 0$, we need only consider that part of $z^{1/n}$ which lies to the right of the binary point in the division Δ/n. If $\Delta < 0$, however, the division $|\Delta|/n$ must be performed and the sum $1 - |\Delta|/n$ formed. This implies two subtraction operations, and it will be assumed that these must have taken place in order to create a worst-case example.

1). Single multiplication, $z = xa_k$:

maximum execution time = $1M_s + 1M + 2A + (1/2)D + 2MA$

2). Two multiplications, $z = xa_kc_j$:

maximum execution time = $2M_s + 2M + 2A + (1/2)D + 4MA$.

5-8: <u>The Log-Exponential Method</u>

The ln x and e^x functions mechanized in the variable structure computer of Cantor, Estrin, and Turn operate upon IBM 7090 floating-point words (8-bit exponent, 27-bit fraction, and sign) and it is for such operands that the execution times will be given. Two timings are given, one for maximum parallelism and the other for a sequential computation.

1). ln x :

$$\text{Parallel} = 1MA + 2M_s + 1A + 1N \;;$$

$$\text{Sequential} = 2MA + 2M_s + 3A + 1N \;;$$

2). e^x :

$$\text{Parallel} = 1C + 1MA + 2M_s + 3A + 1N \;;$$

$$\text{Sequential} = 1C + 3MA + 2M_s + 4A + 1N,$$

where N = normalization and C = conversion. The normalization and conversion consist of a controlled sequence of one bit-position shifts. The normalization takes a minimum of 0 and a maximum of 27 shifts, and the conversion a minimum of 0 and a maximum of 26 shifts. It is seen that the difference between the parallel and sequential computations for ln x is 1MA + 2A, and for e^x, 2MA + 1A. In order to determine the total time needed to compute $x^{1/n}$, the individual computations must be cascaded into the sequence shown in Figure 4-1. Since the difference in computation time between the log-exponential employing parallel and sequential ln x

and e^x is only 3MA + 3A, the sequential methods will be considered. These are the algorithms executed by the variable structure computer designed by Cantor, Estrin, and Turn. The total computation time for the log-exponential n^{th} root is a maximum of $5MA + 4M_s + 7A + 80S$. This time is for the combined $\ln x$-e^x structure [2] employing 1037 stored constants.

CHAPTER VI

Conclusion

The component terms in the maximum expected execution times, in terms of the basic arithmetic and logical operations previously set forth, are given in Table 6-1 for the workable n^{th} rooting methods.

In some instances it may be advantageous to combine two of the previously described methods in a sequential fashion to obtain an advantage in speed. One such example is the use of the Euler iteration formulae plus an initial approximation. When applying the Euler iteration formulae it is common practice in programming, and indeed desirable, to lead into the iterations with a good approximation to the desired root, thus minimizing the number of time-consuming iterations required for full precision. The only iteration formula worthy of consideration in view of the STL log-exponential method is the Newton - Raphson formula, I_{00}. This is a second-order formula, i.e., if a reasonably close approximation is obtained, the error is approximately squared with each succeeding iteration. For example, if we use a Pade approximation to an error $|\epsilon| \leq 2^{-14}$ (relative error $= 2^{-13}$), and apply one Newton - Raphson iteration to this initial value, the result will be within the error bound 2^{-27}. The computation time will be $3MA + 2A + 1D$ for the

Method		MA	A	M	M_s	D	S	S*
STL log-exponential		5	7	0	4	0	80	0
N. R. Binomial Theorem; n > 2		0	54n-54	0	81n-135	D(8)	n+26	54n-81
Euler Formulae (per iteration)	I_{00}	0	1	n-1	0	1	0	0
	I_{10}	0	3	n+3	0	1	0	0
	I_{01}	0	2	2n-1	4	1	0	0
	I_{20}	0	4	n+6	0	1	0	0
	I_{11}	0	4	n+1	4	2	0	0
	I_{02}	0	4	2n+3	6	1	0	0
Padé (first order)		3	2	0	0	1+D(8)	n-1	0
Truncated Series: 1 mult.		1	2	0	1	$\frac{1}{2}$+D(8)	n-1	0
2 mult.		4	2	2	2	$\frac{1}{2}$+D(8)	n-1	0

Table 6-1: Summary of the Terms of the Maximum Execution Times for the Usable nth Rooting Methods, Expressed in Arithmetic and Logical Operations, Single Precision IBM 7090 Floating-Point Operands.

Pade approximation, plus $1A + (n-1)M + 1D$ for the Newton-Raphson iteration, plus $(n-1)S + D(8)$ to reckon the exponent, making a total of $3MA + 3A + (n-1)M + 1D + D(8) + (n-1)S$.

The Padé approximation and truncated series mechanizations are organizationally similar to that of the STL log - exponential method, and are given in Figures 6-1 and 6-2. The micro flow charts for these methods (mantissa part) are given in Figures 6-3 and 6-4. Both the mechanization and micro flow charts for the STL log-exponential method are given in the report by Cantor, Estrin and Turn [2].

The timing evaluations of the various methods were given as sums of multiples of the basic arithmetic and logical operations. In order to directly compare one method with another, a more common time base must be specified. One way of doing this is to designate one of the basic operations as a unit time, and then express the remaining operations as multiples of this time unit, giving all execution times in terms of the time unit.

As an example, if we were to choose the IBM 7090 operation timings, using the one bit-position shift as our time unit, we would obtain the ratios given in Table 6-2. A ono bit position shift in the IBM 7090 takes 1/12 of a 2.18 microsecond machine cycle, or 0.183 microseconds.

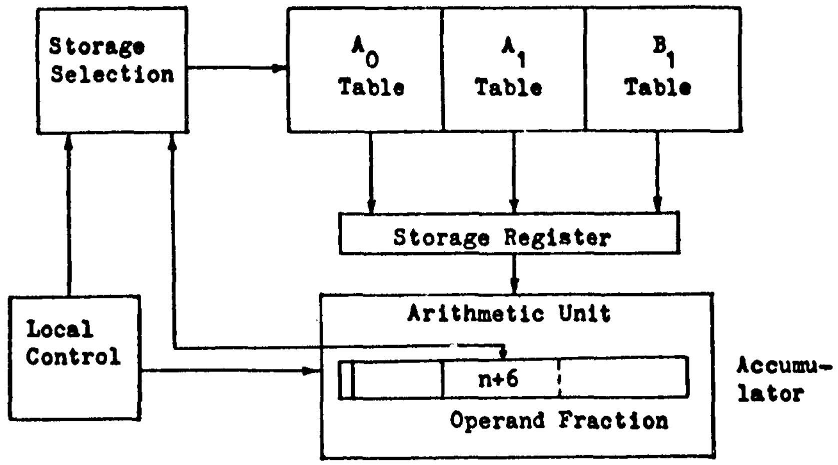


Figure 6-1: Pade Approximation; Organization.

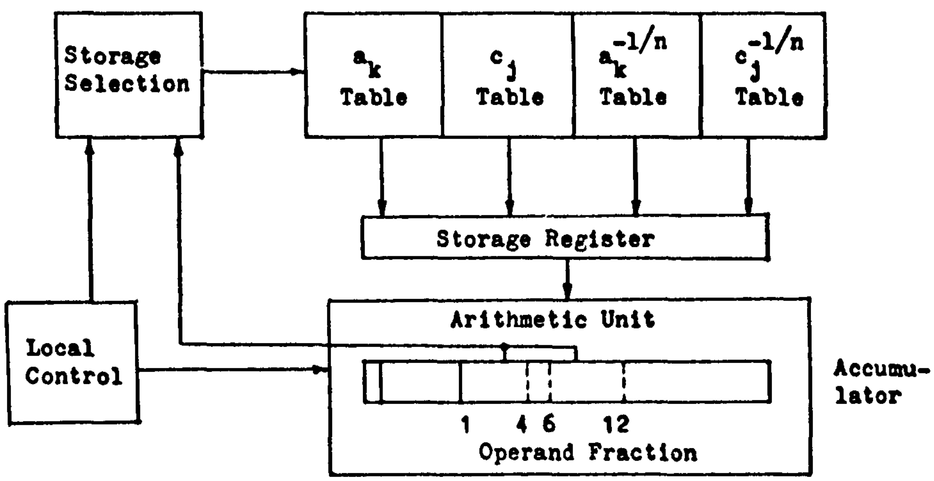


Figure 6-2: Truncated Series; Organization.

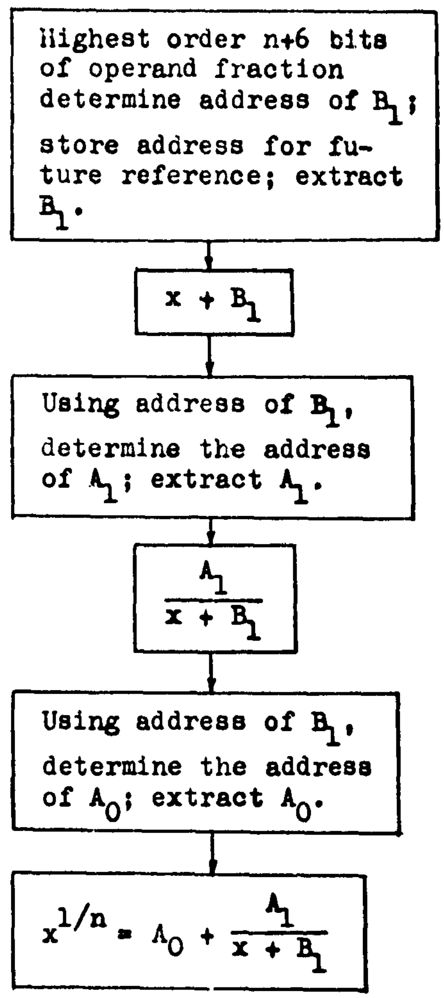


Figure 6-3: Micro Flow Chart for Padé Approximation, n Fixed, Maximum Relative Error $1.5 \cdot 10^{-8}$.

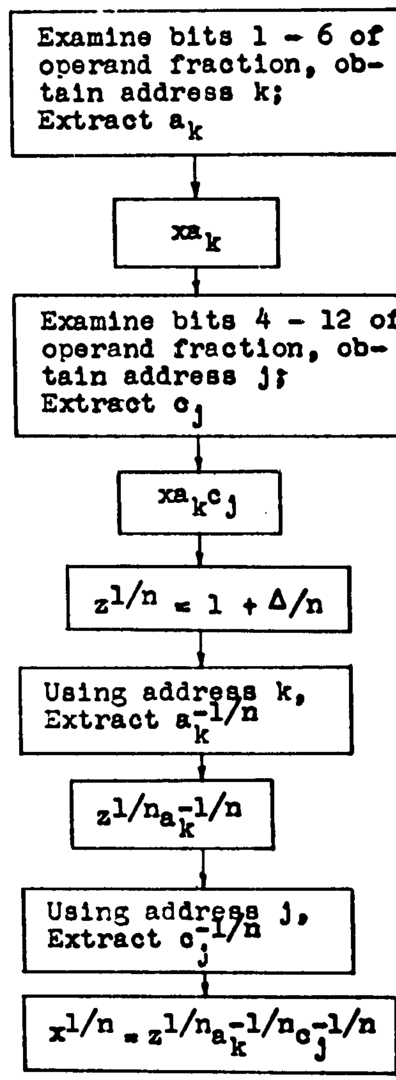


Figure 6-4: Micro Flow Chart for Truncated Series Method, Two Multiplication Scheme, n Fixed.

Op.	Min.	Avg.	Max.
S.	1	1	1
MA	6	6	6
A	6	6	6
M	0	86	108
M_s	0	29-43	36-54
D	0	108	108

Table 6-2: Operation Ratios for Fixed-Point Arithmetic and Logic in the IBM 7090 Arithmetic Unit.

In the above table, all operations are fixed-point binary, with a full-word length of 27 bits. Operand fetch and operation decoding were not included in the above timings. The short multiplication, M_s, may be anywhere from 1/3 to 1/2 the length of a full-word multiplication, depending upon the method in which it is used. If the maximum operation ratios are substituted into the timing expressions in Table 6-1, the value of n at which each method becomes as time-consuming as the STL log-exponential method may be estimated. The key to Table 6-3 is:

(+) No crossover; takes longer than ln-exp.

(-);k No crossover for reasonable size n; Takes less than ln-exp. k = approx. fraction of ln-exp. time.

* Per iteration

Method	Timing Cross-over with STL log-exp. method.
Binomial Theorem: $n = 2$	(-); 0.2
$n > 2$	$n > 2$
Euler Formulae: * I_{00}	$n > 4$
I_{10}	(+)
I_{01}	(+)
I_{20}	(+)
I_{11}	(+)
I_{02}	(+)
Padé Approximation	(-); 0.3
Truncated Series: 1 mult.	(-); 0.5
2 mult.	(-); 0.9
Padé - one I_{00}	$n > 2$

Table 6-3: Timing Crossover Points for the n^{th} Rooting Methods.

The stored table requirements of those methods which require stored constants are summarized in Table 6-4.

Method	Approx. Table Size for small values of n	Table Size Crossover with STL ln-exp. method
STL ln-exp.	1037*	---
Truncated Series: 1 mult. ($n = 2,3,4,5$)	12,000-16,000	(+);12-16
2 mult.	288-316	(-);0.3
Padé Approximation ($n = 2,3,4,5,6,7$)	354-1050	$n > 6$

Table 6-4: Stored Constant table Size Crossover Points for the n^{th} Rooting Methods.

Key: (+);k no crossover; greater than ln-exp. k=ratio.
(-);k no crossover; less than ln-exp. k= ratio.
* independent of n.

Conclusions

Of all the n^{th} rooting methods examined, the STL log-exponential method has been found to be the most versatile, and in most cases the fastest. Traub reports a similar conclusion in his comparison of programmed iterative methods for the n^{th} roots [12] versus use of $\ln x$ and e^x subroutines.

For the special case of the square root the binomial theorem method is desirable from both the timing and mechanization viewpoints. In fact, the square root could be incorporated in a conventional arithmetic unit with the addition of some logical circuitry because of its close relationship to the division operation. The nonrestoring square rooting method has been found to have a time advantage over the related restoring method, as was borne out by the simulation.

For the higher roots, the Padé approximation and the truncated series methods are faster than the log-exponential method. Both methods require tables of stored constants corresponding to each value of n, the truncated series method requiring a lesser number of constants. However, the truncated series method encounters difficulties when the operand is near the interval end-point 2^{-n} when n gets large, whereas the Pade approximation has no such difficulties, and thus the latter is

preferable when n is large.

The Euler iteration formulae are entirely too time consuming to be mechanized because of the superiority of other available methods. Extensions of Nadler's method defy reasonable mechanization, and thus are not useful.

It is recommended that the nonrestoring version of the binomial theorem method be used for the square root. For higher roots, the Padé approximation or the truncated series methods should be used if the problem in question is sufficiently specialized to require a large number of n^{th} roots for fixed n. Otherwise, for the sake of maximum versatility per unit equipment expenditure, it is recommended that the STL log-exponential method be used.

Among other procedures which might well be considered in further study of this problem are those making use of unconventional number representations.

BIBLIOGRAPHY

1. Bemer, R., "A Subroutine Method For Calculating Logarithms", Comm. ACM 1: 5-7, May, 1958.

2. Cantor, D., Estrin, G., and Turn, R., "Computation of Elementary Functions, ln x and e^x, in a Variable Structure Computer", University of California, Los Angeles, Dept. of Engineering, Tech. Report no. 61-16, 1961.

3. Coveyou, R. R., "Serial Correlation in the Generation of Pseudo-Random Numbers", Jour. ACM 7: 72-74, January, 1960.

4. Freiman, C. V., "Statistical Analysis of Certain Binary Division Algorithms", Proc. IRE 49: 91-103, January, 1961.

5. Hald, A., Statistical Theory With Engineering Applications, Wiley, New York, 1952, Chapter 7.

6. International Business Machines Corporation, IBM Customer Engineering Manual For the 7090 Data Processing System, Vol. 1, Form 223-6860-1, 1959.

7. Nadler, M., "Division By the Method of Radixes in Computing Machines", Stroje na zpracovani informaci (Praha) 4: 79-102, 1956 (in Czech).

8. Nadler, M., "Division and Square Root in the Quater-Imaginary Number System", Comm. ACM 4: 192-193, April, 1961.

9. Ralston, A., and Wilf, H., Ed., Mathematical Methods For Digital Computers, Van Nostrand, New York, 1960, Chapter 1.

10. Richards, R. K., Arithmetic Operations in Digital Computers, Van Nostrand, New York, 1955, pp. 279-282.

11. Rotenberg, A., "A New Pseudo-Random Number Generator", Jour. ACM 7: 75-77, January, 1960.

12. Traub, J. F., "Comparison of Iterative Methods For the Calculation of Nth Roots", Comm. ACM 4: 143-145, March, 1961.

13. Traub, J. F., "On a Class of Iteration Formulas and Some Historical Notes", Comm. ACM 4: 276-278, June, 1961.

APPENDIX

Programs for the Property Distribution

MAIN: Calls the input and initialization routine, generates the pseudo-random operands, and takes their square root one at a time, calls the subtotaling and output routines every 1024 operands. Flow diagram given in Fig. A-1.

INPUT: Essential duty is to set to zero all the data areas before performing the experiment.

RT(2): Binary square root simulation program. Contains counters that count up number of iterations, normalizing shifts, and corrections for each operand. Flow diagram given in Fig. 3-6 .

PFXSRT: Identifies the range of each operand by comparing it against a table (PFXTBL), placing an address modifier in index register 1 so that the results may be determined versus operand magnitude.

SUBTOT: Takes the tally of the fixed-point counters, converts them to floating point, and computes the output information.

RBIT = root bits per iteration;

PSHFT = shifts per operand;

PXITER = iterations per operand;

PCORR = corrections per operand;

PFREQ = relative frequency of operands.

OUTPUT: Contains the output formats. Prints out the

quantities computed by SUBTOT every 1024 operands.

```
START SWT    1              CHANGE INITIAL RANDOM NUMBER IF DOWN
      TRA    EXPT           NO CHANGE
CHNG  HPR    32767          FOR MANUAL DATUM ENTRY
      ENK                   NEW DATUM INTO MQ REGISTER
      HPR    32767          TURN OFF SENSE SWITCH 1
      XCA                   NEW DATUM INTO AC
      TZE    CHNG           PRECAUTION AGAINST ENTERING ZERO
      STO    MPCX           STORE NEW INITIAL RANDOM NUMBER
      STO    RANDOM
      REM    BEGIN EXPERIMENT
EXPT  NOP
      CLA    MPCX           FOR OUTPUT
      STO    RANDOM
      STZ    JGROUP         CLEAR GROUP COUNTER
      CALL   INPUT          INPUT AND INITIALIZATION
      AXT    16,2
      REM    CONTINUE EXPERIMENT
CONT  NOP
      REM    CLEAR DATA AREAS
      AXT    32,1
      STZ    ISHFT+1,1      CLEAR SHIFT COUNTERS
      STZ    ITER+1,1       CLEAR ITERATION COUNTERS
      STZ    ICORR+1,1      CLEAR RESTORATION COUNTERS
      STZ    IFREQ+1,1      CLEAR OPERAND DISTRIBUTION
      TIX    *-4,1,1
      STZ    IERROR         CLEAR ERROR FAILURE COUNTER
      STZ    ICHECK         CLEAR CHECK FAILURE COUNTER
      CLA    JGROUP         UPDATE GROUP COUNTER
      ADD    INT1
      STD    JGROUP
```

```
        CALL    TATTLE,JGROUP VISUAL DISPLAY OF GROUP COUNTER
        REM     PERFORM EXPERIMENTS
        AXT     1024,1
        TSX     MPC,4
        CALL    RT(2),X         EXTRACT SQUARE ROOT
        TIX     *-3,1,1
        CALL    SUBTOT          SUBTOTAL RESULTS
        CALL    OUTPUT          OUTPUT RESULTS
OPTNS   CALL    SAVE            'SAVE' OPTION, SENSE SWITCH 5
        TIX     CONT,2,1
        REM     EXPERIMENT OPTIONS
        SWT     1               CHANGE INITIAL RANDOM NUMBER IF DOWN
        TRA     *+2             NO CHANGE
        TRA     START           BACK TO THE VERY BEGINNING
        CALL    EXIT            SIGN OFF
JGROUP  BSS     1               GROUPS OF 'XNUM' COUNTER
INT1    PZE     ,,1             FORTRAN INTEGER 1
        REM     MULTIPLICATIVE CONGRUENCE GENERATOR
MPC     NOP
        CLA     MPCX            FORM PSEUDO RANDOM NUMBER
        ADD     MPCC
        STO     MPCX
        SUB     MPCC
        ALS     11
        ADD     MPCX
        LRS     26              MODULO, P=26
        CLM
        LLS     26
        STO     MPCX            SAVE FOR NEXT GENERATION
        ALS     8               POSITION
```

```
        ORA     MPCLBT          FORCE FIRST BIT TO BE 1
        ARS     1               **********                                  ********
        STO     X               PSEUDO RANDOM OPERAND
        TOV     *+1             RESET OVERFLOW INDICATOR
        TRA     1,4             RETURN
 MPCX   OCT     000232544614    PREVIOUS PSEUDO RANDOM NUMBER
 MPCC   OCT     000000000001    ALGORITHM CONSTANT
MPCLBT  PTW                     LEADING BIT
 X      BSS     1               OPERAND
*
RANDOM  COMMON  1               INITIAL RANDOM NO. FOR MPC
 N      COMMON  1               ROOT ORDER
IERROR  COMMON  1               ERROR COUNTER
ICHECK  COMMON  1               CHECK FAILURE COUNTER
 ISHFT  COMMON  32              SHIFT COUNTERS
 ITER   COMMON  32              ITERATION COUNTERS
 ICORR  COMMON  32              RESTORATION COUNTERS
 IFREQ  COMMON  32              OPERAND DISTRIBUTION
 XNUM   COMMON  1               NO. OF OPERANDS
        END
```

Table A-1: Main Program for Property Distribution

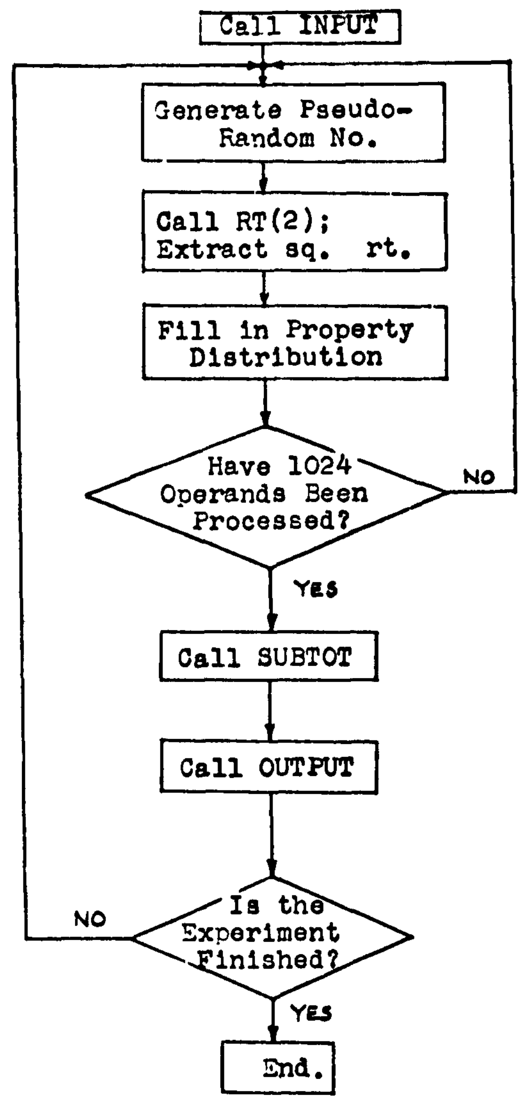


Fig. A-1: Flow Chart for Property Distribution Main Program, pp. 119-121 .

```
*       BINARY SQUARE ROOT SIMULATION PROGRAM, PROPERTY DIST.
        ENTRY   RT(2)
 RT(2)  NOP
        SXD     XRSV,1
        SXD     XRSV+1,2
        SXD     XRSV+2,4
        REM     INITIALIZE ALL REGISTERS
        CLA     =1
        ALS     32
        SLW     XSQ             INITIALIZE LOW-ORDER SQUARE REGISTER
        SLW     TFR             INITIALIZE TRIAL FACTOR REGISTER
        ALS     1
        SLW     DGLINE          INITIALIZE CURRENT DIGIT
        CLA*    1,4             BRING IN OPERAND
        STO     OPR
        ARS     1               CORRECT POSITIONING
        STO     REMR            INITIALIZE REMAINDER REGISTER
        STZ     IISHFT          CLEAR SHIFT SUB-COUNTER
        STZ     IITER           CLEAR ITERATION SUB-COUNTER
        STZ     IICORR          CLEAR RESTORATION SUB-COUNTER
        SLF                     TURN OFF THE SENSE LIGHTS
        LDI     ENDBIT          END MARKER INTO SENSE INDICATORS
        REM     IDENTIFY OPERAND PREFIX
        CALL    PFXSRT,OPR      IDENTIFY OPERAND PREFIX
        CLA     IFREQ+1,1       FILL IN OPERAND DISTRIBUTION
        ADD     INT1
        STD     IFREQ+1,1
        REM     PERFORM AN ITERATION
 NEXT   CLA     IITER           UPDATE ITERATION SUB-COUNTER
        ADD     INT1
```

```
        STD     IITER
        REM     CHECK SIGN OF REMR
        CLA     REMR
        TMI     REMNEG
        REM     THE REMAINDER WAS POSITIVE
REMPOS  SUB     TFR            SUBTRACT TRIAL FACTOR
        ALS     1              ADJUST NEW REMAINDER
        TRA     NEWREM         EXAMINE NEW REMAINDER
        REM     THE REMAINDER WAS NEGATIVE
REMNEG  ADD     TFR            ADD TRIAL FACTOR
        ALS     1              ADJUST NEW REMAINDER
        TMI     ERROR          IMPOSSIBLE
        TZE     EXACT          EXACT ROOT
        TRA     NEWREM         EXAMINE NEW REMAINDER
 EXACT  SSP                    CANCEL A POSSIBLE NEGATIVE ZERO
        REM     CHECK SIGN OF NEW REMAINDER
        REM     SAVE NEW REMAINDER, UPDATE, ETC.
NEWREM  STO     REMR           SAVE REMAINDER
        TPL     *+2            (+), SLF
        SLN     1              (-), SLN
        CAL     XSQ            MOVE LOW-ORDER SQUARE
        ARS     1
        SLW     XSQ            STORE NEW LOW-ORDER SQUARE
        CAL     DGLINE         INJECT CURRENT DIGIT
        SLT     1              TEST (+) OR (-)
        ORS     TFR            1 TO PARTIAL ROOT
        ARS     1              MOVE CURRENT DIGIT
        TIO     CHECK          TEST TO SEE IF FINISHED
        SLW     DGLINE         STORE NEW CURRENT DIGIT
        CAL     TFR            MODIFY TRIAL FACTOR
```

```
        ERA     DGLINE        ERASE OLD LOW-ORDER SQUARE
        ORA     XSQ           INJECT NEW LOW-ORDER SQUARE
        SLW     TFR           STORE NEW TRIAL FACTOR
        TRA     LZ
        REM     CHECK FOR LEADING ZEROS
LZ      CLA     REMR          SET UP REMAINDER
        TMI     LZ)N
LZ)P    TSX     BT,4          1-BIT TEST
        PZE     0,2
        TRA     NLZ           SKIP LEADING ZERO TEST
        TRA     LZA+1         JUMP INTO SHIFT LOOP
LZ)N    TSX     BT,4          2-BIT TEST
        PZE     0,1
        TRA     LZB+1         SKIP LEADING ZERO TEST, CORRECT REMR
        TRA     LZA+1         JUMP INTO SHIFT LOOP
LZA     CLA     REMR          RELOAD REMAINDER INTO AC
        TMI     )N
)P      TSX     BT,4          TEST FOR LEADING ZERO
        PZE     0,1           REMR (+), 2-BIT TEST
        TRA     LZB           = 1
        TRA     UPD           = 0, SHIFT OUT A LEADING ZERO
)N      TSX     BT,4          TEST FOR LEADING ZERO
        PZE     16384,0       REMR (-), 3-BIT TEST
        TRA     LZB           = 1
UPD     ALS     1             = 0, SHIFT OUT A LEADING ZERO
        STO     REMR          SAVE REMAINDER
        TMI     *+2           SENSE LIGHT 1 ON IF REMR IS (+)
        SLN     1             TURN ON THE SENSE LIGHT
        CLA     IISHFT        UPDATE SHIFT SUB-COUNTER
        ADD     INT1
```

```
        STD    IISHFT
        CAL    XSQ          MOVE LOW-ORDER SQUARE
        ARS    1
        SLW    XSQ          STORE NEW LOW-ORDER SQUARE
        CAL    DGLINE       INJECT CURRENT DIGIT
        SLT    1            TEST SENSE LIGHT 1
        ORS    TFR          REMR (-), 1 TO PARTIAL ROOT
        ARS    1            REMR (+), MOVE CURRENT DIGIT
        TIO    CHECK        TEST TO SEE IF FINISHED
        SLW    DGLINE       STORE NEW CURRENT DIGIT
        CAL    TFR          MODIFY TRIAL FACTOR
        ERA    DGLINE       ERASE OLD LOW-ORDER SQUARE
        ORA    XSQ          INJECT NEW LOW-ORDER SQUARE
        SLW    TFR          STORE NEW TRIAL FACTOR
        TRA    LZA          TRY AGAIN
LZB     TPL    NCORR        NO MORE LEADING ZEROS
        ADD    DGLINE       CORRECTION IF (-)
NCORR   STO    REMR
        TPL    NEXT         NEXT ITERATION IF (+)
        CLA    IICORR       UPDATE RESTORATION SUB-COUNTER
        ADD    INT1
        STD    IICORR
NXT     TRA    NEXT         PERFORM NEXT ITERATION
NLZ     STO    REMR
        TRA    NEXT         PERFORM NEXT ITERATION
ERROR   CLA    IERROR       UPDATE ERROR COUNTER
        ADD    INT1
        STO    IERROR
        TRA    RSTR         STATUS QUO
NCHECK  CLA    ICHECK       UPDATE CHECK COUNTER
```

```
       ADD     INT1
       STO     ICHECK
       TRA     RSTR           STATUS QUO
       REM     CHECK THE RESULT
CHECK  CLA     TFR            MAKE ALLOWANCE FOR SHORT REGISTER
       ANA     MSK            WIPE OUT EXCESS POSITIONS
       ALS     1              NORMALIZE FOR TEST PURPOSES
       STO     ROOT           FOR MULTIPLICATION
       XCA                    FOR MULTIPLICATION
       MPY     ROOT           SQUARE THE RESULT
       LRS     8              ROUND OFF
       RND
       ALS     8              REPOSITION
       STO     RTSQ           FOR COMPARISON
       SUB     OPR            OBTAIN DIFFERENCE
       ARS     10             SHIFT OUT MINIMUM ACCEPTABLE DIFFERENCE
       TNZ     NCHECK         DIFFERENCE TOO LARGE
       REM     FINISHED WITH THIS OPERAND
FINIS  CLA     ITER+1,1       SUBTOTAL ITERATION COUNTER
       ADD     IITER
       STD     ITER+1,1
       CLA     ISHFT+1,1      SUBTOTAL SHIFT COUNTER
       ADD     IISHFT
       STD     ISHFT+1,1
       CLA     ICORR+1,1      SUBTOTAL RESTORATION COUNTER
       ADD     IICORR
       STD     ICORR+1,1
RSTR   LXD     XRSV,1
       LXD     XRSV+1,2
       LXD     XRSV+2,4
```

```
        TOV    *+1
        TRA    2,4             RETURN
        REM    LEFT HALF BIT TEST ROUTINE
BT      STI    BTSV            SAVE INDICATORS
        STO    REMSV           SAVE REMAINDER
        PAI                    REMR TO INDICATORS
        CLA    1,4             SET UP INDICATOR TEST
        STT    *+2
        STA    *+1
        LNT    0               ON TEST FOR LEADING BIT
        SLN    2               = 0
        CLA    REMSV           = 1, REMR TO AC
        LDI    BTSV            RESTORE INDICATORS
        SLT    2               TEST FOR ON OR OFF
        TRA    2,4             = 1
        TRA    3,4             = 0
BTSV    BSS    1               SAVED INDICATORS
REMSV   BSS    1               SAVED REMAINDER
MSK     OCT    377777777600    MASK TO MAKE SHORT REGISTER
OPR     BSS    1               OPERAND FOR ANSWER CHECK
REMR    BSS    1               REMAINDER REGISTER (AUGEND)
TFR     BSS    1               TRIAL FACTOR REGISTER (ADDEND)
DGLINE  BSS    1               DIGIT LINES
XSQ     BSS    1               LOW-ORDER SQUARE REGISTER
ROOT    BSS    1               NORMALIZED ROOT FOR ANSWER CHECK
RTSQ    BSS    1               SQUARED SQUARE ROOT
ENDBIT  OCT    000000000100    END MARKER
INT1    PZE    ,,1             FORTRAN INTEGER 1
XRSV    BSS    3               INDEX REGISTERS
IISHFT  BSS    1               SHIFT SUB-COUNTER
```

```
 IITER BSS       1              ITERATION SUB-COUNTER
IICORR BSS       1              RESTORATION SUB-COUNTER
*
RANDOM COMMON    1              INITIAL RANDOM NO. FOR MPC
 N     COMMON    1              ROOT ORDER
IERROR COMMON    1              ERROR COUNTER
ICHECK COMMON    1              CHECK FAILURE COUNTER
 ISHFT COMMON    32             SHIFT COUNTERS
 ITER  COMMON    32             ITERATION COUNTERS
 ICORR COMMON    32             RESTORATION COUNTERS
 IFREQ COMMON    32             OPERAND DISTRIBUTION
 XNUM  COMMON    1              NO. OF OPERANDS
       END
```

Table A-2: Binary Square Root Simulation Program, Property Distribution.

```
*       PREFIX IDENTIFYING ROUTINE
        ENTRY   PFXSRT
PFXSRT  NOP
        REM     CORRECT ADDRESS MODIFIER IS PLACED IN X.R. 1
        STI     INDS            SAVE INDICATORS
        LDI*    1,4             OPERAND TO INDICATORS
        AXT     32,1
        ONT     PFXTBL+1,1      MATCHING TEST
        TIX     *-1,1,1
        LDI     INDS            RESTORE INDICATORS
        TRA     2,4             RETURN
        REM     PREFIX TABLE
        OCT     176000000000,174000000000
        OCT     172000000000,170000000000
        OCT     166000000000,164000000000
        OCT     162000000000,160000000000
        OCT     156000000000,154000000000
        OCT     152000000000,150000000000
        OCT     146000000000,144000000000
        OCT     142000000000,140000000000
        OCT     136000000000,134000000000
        OCT     132000000000,130000000000
        OCT     126000000000,124000000000
        OCT     122000000000,120000000000
        OCT     116000000000,114000000000
        OCT     112000000000,110000000000
        OCT     106000000000,104000000000
        OCT     102000000000
PFXTBL  OCT     100000000000
 INDS   BSS     1               SAVED SENSE INDICATORS
        END
```

Table A-3: Prefix Identifying Routine, $0.25 \leq x < 0.5$.

```
*       PREFIX IDENTIFYING ROUTINE
        ENTRY   PFXSRT
PFXSRT  NOP
        REM     CORRECT ADDRESS MODIFIER IS PLACED IN X.R. 1
        STI     INDS            SAVE INDICATORS
        LDI*    1,4             OPERAND TO INDICATORS.
        AXT     32,1
        ONT     PFXTBL+1,1      MATCHING TEST
        TIX     *-1,1,1
        LDI     INDS            RESTORE INDICATORS
        TRA     2,4             RETURN
        REM     PREFIX TABLE
        OCT     374000000000,370000000000
        OCT     364000000000,360000000000
        OCT     354000000000,350000000000
        OCT     344000000000,340000000000
        OCT     334000000000,330000000000
        OCT     324000000000,320000000000
        OCT     314000000000,310000000000
        OCT     304000000000,300000000000
        OCT     274000000000,270000000000
        OCT     264000000000,260000000000
        OCT     254000000000,250000000000
        OCT     244000000000,240000000000
        OCT     234000000000,230000000000
        OCT     224000000000,220000000000
        OCT     214000000000,210000000000
        OCT     204000000000
PFXTBL  OCT     200000000000
 INDS   BSS     1               SAVED SENSE INDICATORS
        END
```

Table A-4: Prefix Identifying Routine, $0.5 \leq x < 1$.

```
      SUBROUTINE INPUT
C
C     INPUT AND INITIALIZATION ROUTINE
C
      DIMENSION DUMMY1(130),DUMMY2(160)
      DIMENSION SHFT(32),XITER(32),CORR(32),FREQ(32)
      COMMON RANDOM
      COMMON N
      COMMON DUMMY1
      COMMON XNUM,XNUMB
      COMMON DUMMY2
      COMMON SHFT,XITER,CORR,FREQ
    3 FORMAT(I2,F10.0)
      READ INPUT TAPE 5,3,N,XNUM
      XNUMB=0.0
      DO 10 I=1,32
      SHFT(I)=0.0
      XITER(I)=0.0
      CORR(I)=0.0
      FREQ(I)=0.0
   10 CONTINUE
      RETURN
      END
```

Table A-5: Input Routine, Property Distribution.

```
      SUBROUTINE SUBTOT
C     SUBTOTALING ROUTINE
      DIMENSION ISHFT(32),ITER(32),ICORR(32),IFREQ(32)
      DIMENSION RBIT(32),PSHFT(32),PXITER(32),PCORR(32),PFREQ(32)
      DIMENSION XITER(32),SHFT(32),CORR(32),FREQ(32)
      DIMENSION XSHFT(32),XXITER(32),XCORR(32),XFREQ(32)
      COMMON RANDOM
      COMMON N,IERROR,ICHECK,ISHFT,ITER,ICORR,IFREQ,XNUM,XNUMB
      COMMON RBIT,PSHFT,PXITER,PCORR,PFREQ
      COMMON SHFT,XITER,CORR,FREQ
      XNUMB=XNUMB+XNUM
      DO 10 I=1,32
      XSHFT(I)=ISHFT(I)
      SHFT(I)=SHFT(I)+XSHFT(I)
      XXITER(I)=ITER(I)
      XITER(I)=XITER(I)+XXITER(I)
      XCORR(I)=ICORR(I)
      CORR(I)=CORR(I)+XCORR(I)
      XFREQ(I)=IFREQ(I)
      FREQ(I)=FREQ(I)+XFREQ(I)
      RBIT(I)=(SHFT(I)+XITER(I))/XITER(I)
      PSHFT(I)=SHFT(I)/FREQ(I)
      PXITER(I)=XITER(I)/FREQ(I)
      PCORR(I)=CORR(I)/FREQ(I)
      PFREQ(I)=FREQ(I)/XNUMB
   10 CONTINUE
      RETURN
      END
```

Table A-6: Subtotaling Routine, Property Distribution.

```
      SUBROUTINE OUTPUT
C     OUTPUT ROUTINE
      DIMENSION DUMMY(128)
      DIMENSION RBIT(32),PSHFT(32),PXITER(32),PCORR(32),PFREQ(32)
      COMMON RANDOM
      COMMON N,IERROR,ICHECK
      COMMON DUMMY
      COMMON XNUM,XNUMB
      COMMON RBIT,PSHFT,PXITER,PCORR,PFREQ
  100 FORMAT(44H1     RESULTS OF BINARY ROOT SIMULATION, N =I2)
  101 FORMAT(48H0     NUMBER OF OPERANDS SUCCESSFULLY PROCESSED F10.0)
  102 FORMAT(4H    )
  103 FORMAT(22H0                OPERAND)
  104 FORMAT(97H                   PREFIX      100000    100001    100010    1
     X00011    100100    100101    100110    100111)
  105 FORMAT(97H                   PREFIX      101000    101001    101010    1
     X01011    101100    101101    101110    101111)
  106 FORMAT(97H                   PREFIX      110000    110001    110010    1
     X10011    110100    110101    110110    110111)
  107 FORMAT(97H                   PREFIX      111000    111001    111010    1
     X11011    111100    111101    111110    111111)
  108 FORMAT(19H0     ROOT BITS PER)
  109 FORMAT(25H      ITERATION          8F9.2)
  110 FORMAT(20H0     ITERATIONS PER)
  111 FORMAT(25H      OPERAND            8F9.2)
  112 FORMAT(16H0     SHIFTS PER)
  113 FORMAT(22H0     RESTORATIONS PER)
  114 FORMAT(20H0     RELATIVE FREQ.)
  115 FORMAT(25H      OF OPERANDS        8F9.3)
  116 FORMAT(35H0     ERROR FAILURES IN THIS GROUP I5)
```

```
117  FORMAT(35H      CHECK FAILURES IN THIS GROUP I5)
118  FORMAT(61H0     INITIAL RANDOM NUMBER USED BY RANDOM NUMBER GENERA
    1TOR  O12)
     WRITE OUTPUT TAPE 6,100,N
     WRITE OUTPUT TAPE 6,118,RANDOM
     WRITE OUTPUT TAPE 6,101,XNUMB
     WRITE OUTPUT TAPE 6,102
     WRITE OUTPUT TAPE 6,102
     WRITE OUTPUT TAPE 6,103
     WRITE OUTPUT TAPE 6,104
     WRITE OUTPUT TAPE 6,114
     WRITE OUTPUT TAPE 6,115,(PFREQ(I), I= 1, 8)
     WRITE OUTPUT TAPE 6,110
     WRITE OUTPUT TAPE 6,111,(PXITER(I), I= 1, 8)
     WRITE OUTPUT TAPE 6,112
     WRITE OUTPUT TAPE 6,111,(PSHFT(I), I= 1, 8)
     WRITE OUTPUT TAPE 6,113
     WRITE OUTPUT TAPE 6,111,(PCORR(I), I= 1, 8)
     WRITE OUTPUT TAPE 6,108
     WRITE OUTPUT TAPE 6,109,(RBIT(I), I= 1, 8)
     WRITE OUTPUT TAPE 6,102
     WRITE OUTPUT TAPE 6,103
     WRITE OUTPUT TAPE 6,105
     WRITE OUTPUT TAPE 6,114
     WRITE OUTPUT TAPE 6,115,(PFREQ(I), I= 9,16)
     WRITE OUTPUT TAPE 6,110
     WRITE OUTPUT TAPE 6,111,(PXITER(I), I= 9,16)
     WRITE OUTPUT TAPE 6,112
     WRITE OUTPUT TAPE 6,111,(PSHFT(I), I= 9,16)
     WRITE OUTPUT TAPE 6,113
```

```
WRITE OUTPUT TAPE 6,111,(PCORR(I), I= 9,16)
WRITE OUTPUT TAPE 6,108
WRITE OUTPUT TAPE 6,109,(RBIT(I), I= 9,16)
WRITE OUTPUT TAPE 6,102
WRITE OUTPUT TAPE 6,103
WRITE OUTPUT TAPE 6,106
WRITE OUTPUT TAPE 6,114
WRITE OUTPUT TAPE 6,115,(PFREQ(I), I=17,24)
WRITE OUTPUT TAPE 6,110
WRITE OUTPUT TAPE 6,111,(PXITER(I), I=17,24)
WRITE OUTPUT TAPE 6,112
WRITE OUTPUT TAPE 6,111,(PSHFT(I), I=17,24)
WRITE OUTPUT TAPE 6,113
WRITE OUTPUT TAPE 6,111,(PCORR(I), I=17,24)
WRITE OUTPUT TAPE 6,108
WRITE OUTPUT TAPE 6,109,(RBIT(I), I=17,24)
WRITE OUTPUT TAPE 6,102
WRITE OUTPUT TAPE 6,103
WRITE OUTPUT TAPE 6,107
WRITE OUTPUT TAPE 6,114
WRITE OUTPUT TAPE 6,115,(PFREQ(I), I=25,32)
WRITE OUTPUT TAPE 6,110
WRITE OUTPUT TAPE 6,111,(PXITER(I), I=25,32)
WRITE OUTPUT TAPE 6,112
WRITE OUTPUT TAPE 6,111,(PSHFT(I), I=25,32)
WRITE OUTPUT TAPE 6,113
WRITE OUTPUT TAPE 6,111,(PCORR(I), I=25,32)
WRITE OUTPUT TAPE 6,108
WRITE OUTPUT TAPE 6,109,(RBIT(I), I=25,32)
```

```
WRITE OUTPUT TAPE 6,116,IERROR
WRITE OUTPUT TAPE 6,117,ICHECK
RETURN
END
```

Table A-7: Output Routine, Property Distribution.

Programs for the Timing Distribution

MAIN: Calls the input routine, generates the pseudo-random operands takes their square root one at a time, fills in the timing distribution, and calls the output routine at the end of the experiment. Flow diagram given in Fig. A-2.

INPUT2: Reads in the number of operands to be processed.

RT(2): Binary square root simulation program. Uses index register 2 to count up number of time units required to execute each square root. Flow diagram given in Fig. 3-6A.

OUTPT2: The timing distribution, IQ or JQ, is the timing density function. The normalized cumulative distribution function is computed and placed in XQ. All nonzero entries of JQ are printed out, and all entries of XQ are printed out.

```
K      EQU    0
START  SWT    1              CHANGE INITIAL RANDOM NUMBER IF DOWN
       TRA    EXPT           NO CHANGE
CHNG   HPR    32767          FOR MANUAL DATUM ENTRY
       ENK                   NEW DATUM INTO MQ REGISTER
       HPR    32767          TURN OFF SENSE SWITCH 1
       XCA                   NEW DATUM INTO AC
       TZE    CHNG           PRECAUTION AGAINST ENTERING ZERO
       STO    MPCX           STORE NEW INITIAL RANDOM NUMBER
       STO    RANDOM
       REM    BEGIN EXPERIMENT
EXPT   NOP
       CLA    MPCX           FOR OUTPUT DISPLAY
       STO    RANDOM
       STZ    IERROR         CLEAR ERROR FAILURE COUNTER
       STZ    ICHECK         CLEAR CHECK FAILURE COUNTER
       REM                   CLEAR TIMING DISTRIBUTION
       AXT    500,1
       STZ    IQ+1,1
       TIX    *-1,1,1
       CALL   INPUT2         INPUT
       REM    PERFORM EXPERIMENTS
       CLA    NN             NO. OF OPERANDS
       ARS    1              HALVE IT
       STD    JMP            FOR JUMP INSTRUCTION
       LXD    NN,1
       TSX    MPC,4          GENERATE PSEUDO-RANDOM NUMBER
       CALL   RT(2),X        EXTRACT SQUARE ROOT
       CLA    IQ+1+K,2       AUGMENT TIMING DISTRIBUTION
       ADD    INT1
```

```
        STD     IQ+1+K,2
        TIX     *-6,1,1
        CALL    OUTPT2          OUTPUT RESULTS
        SWT     1               CHANGE INITIAL RANDOM NUMBER IF DOWN
        TRA     *+2             NO CHANGE
        TRA     START           BACK TO THE VERY BEGINNING
        CALL    EXIT            SIGN OFF
        REM     MULTIPLICATIVE CONGRUENCE GENERATOR
MPC     NOP
        CLA     MPCX            FORM PSEUDO RANDOM NUMBER
        ADD     MPCC
        STO     MPCX
        SUB     MPCC
        ALS     11
        ADD     MPCX
        LRS     26              MODULO, P=26
        CLM
        LLS     26
        STO     MPCX            SAVE FOR NEXT GENERATION
        ALS     8               POSITION
        ORA     MPCLBT          FORCE FIRST BIT TO BE 1
JMP     TXL     *+2,1,0         JUMP IF FIRST HALF OF EXPT.
        ARS     1               **********                        ********
        STO     X               PSEUDO RANDOM OPERAND
        TOV     *+1             RESET OVERFLOW INDICATOR
        TRA     1,4             RETURN
MPCX    OCT     000232544614    PREVIOUS PSEUDO RANDOM NUMBER
MPCC    OCT     000000000001    ALGORITHM CONSTANT
MPCLBT  PTW                     LEADING BIT
X       BSS     1               OPERAND
```

```
 INT1   PZE      ,,1            FORTRAN INTEGER 1
*
RANDOM COMMON   1               INITIAL RANDOM NUMBER
IERROR COMMON   1               ERROR COUNTER
ICHECK COMMON   1               CHECK FAILURE COUNTER
 IQ    COMMON   500             TIMING DISTRIBUTION
 NN    COMMON   1               NO. OF OPERANDS
 XNN   COMMON   1               NO. OF OPERANDS
       END
```

Table A-8: Main Program for Timing Distribution.

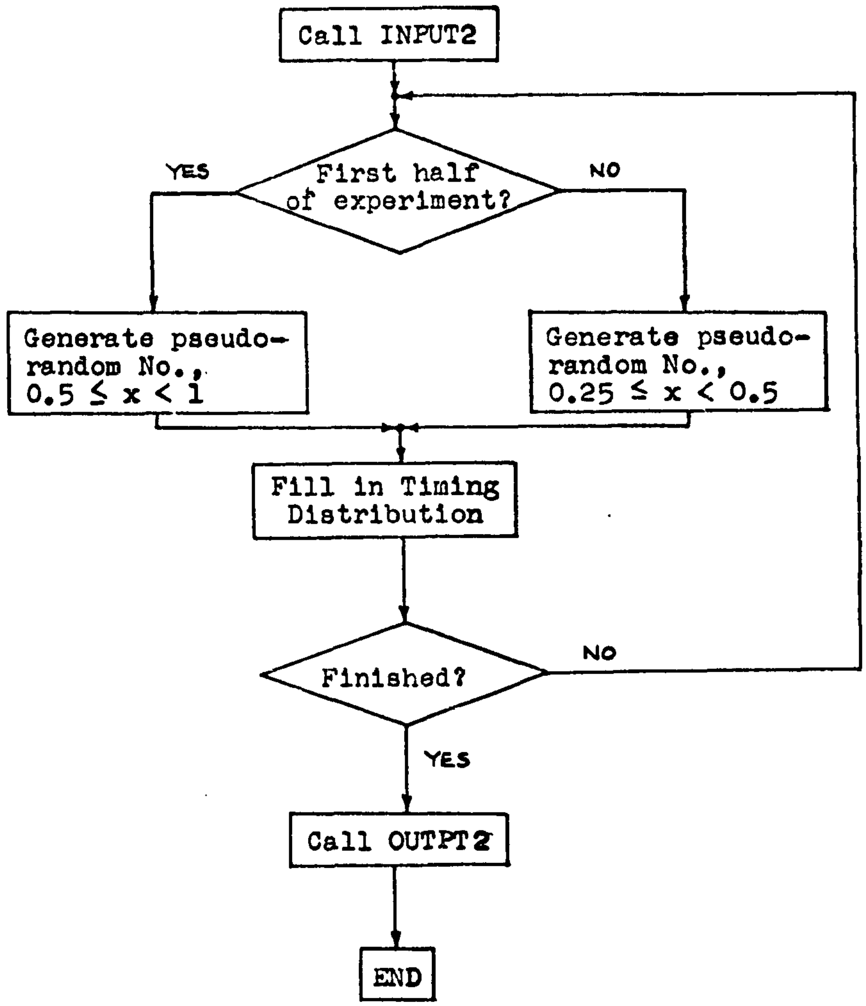


Fig. A-2: Flow chart for Timing Distribution
Main Program, pp. 139-141 .

```
*       BINARY SQUARE ROOT SIMULATION PROGRAM, TIMING DIST.
        ENTRY   RT(2)
TADD    EQU     3                   ADDITION TIME
TA      EQU     1                   AUGMENT AND TRIAL FACTOR
TS      EQU     1                   SHIFT OUT LEADING ZERO
RT(2)   NOP
        SXD     XRSV,1
        SXD     XRSV+2,4
        REM     INITIALIZE ALL REGISTERS
        CLA     =1
        ALS     32
        SLW     XSQ                 INITIALIZE LOW-ORDER SQUARE REGISTER
        SLW     TFR                 INITIALIZE TRIAL FACTOR REGISTER
        ALS     1
        SLW     DGLINE              INITIALIZE CURRENT DIGIT
        CLA*    1,4                 BRING IN OPERAND
        STO     OPR
        ARS     1                   CORRECT POSITIONING
        STO     REMR                INITIALIZE REMAINDER REGISTER
        SLF                         TURN OFF THE SENSE LIGHTS
        LDI     ENDBIT              END MARKER INTO SENSE INDICATORS
        AXT     0,2                 CLEAR TIME UNIT COUNTER
        REM     PERFORM AN ITERATION
NEXT    NOP
        REM     CHECK SIGN OF REMR
        CLA     REMR
        TMI     REMNEG
        REM     THE REMAINDER WAS POSITIVE
REMPOS  SUB     TFR                 SUBTRACT TRIAL FACTOR
        ALS     1                   ADJUST NEW REMAINDER
```

```
        TXI    *+1,2,TADD      UPDATE TIME UNIT COUNTER
        TRA    NEWREM          EXAMINE NEW REMAINDER
        REM    THE REMAINDER WAS NEGATIVE
REMNEG  ADD    TFR             ADD TRIAL FACTOR
        ALS    1               ADJUST NEW REMAINDER
        TXI    *+1,2,TADD      UPDATE TIME UNIT COUNTER
        TMI    ERROR           IMPOSSIBLE
        TZE    EXACT           EXACT ROOT
        TRA    NEWREM          EXAMINE NEW REMAINDER
 EXACT  SSP                    CANCEL A POSSIBLE NEGATIVE ZERO
        REM    CHECK SIGN OF NEW REMAINDER
        REM    SAVE NEW REMAINDER, UPDATE, ETC.
NEWREM  STO    REMR            SAVE REMAINDER
        TPL    *+2             (+), SLF
        SLN    1               (-), SLN
        CAL    XSQ             MOVE LOW-ORDER SQUARE
        ARS    1
        SLW    XSQ             STORE NEW LOW-ORDER SQUARE
        CAL    DGLINE          INJECT CURRENT DIGIT
        SLT    1               TEST (+) OR (-)
        ORS    TFR             1 TO PARTIAL ROOT
        ARS    1               MOVE CURRENT DIGIT
        TIO    CHECK           TEST TO SEE IF FINISHED
        SLW    DGLINE          STORE NEW CURRENT DIGIT
        CAL    TFR             MODIFY TRIAL FACTOR
        ERA    DGLINE          ERASE OLD LOW-ORDER SQUARE
        ORA    XSQ             INJECT NEW LOW-ORDER SQUARE
        SLW    TFR             STORE NEW TRIAL FACTOR
        TXI    *+1,2,TA        UPDATE TIME UNIT COUNTER
        TRA    LZ
```

```
        REM     CHECK FOR LEADING ZEROS
LZ      CLA     REMR          SET UP REMAINDER
        TMI     LZ)N
LZ)P    TSX     BT,4          1-BIT TEST
        PZE     0,2
        TRA     NLZ           SKIP LEADING ZERO TEST
        TRA     LZA+1         JUMP INTO SHIFT LOOP
LZ)N    TSX     BT,4          2-BIT TEST
        PZE     0,1
        TRA     LZB+1         SKIP LEADING ZERO TEST, CORRECT REMR
        TRA     LZA+1         JUMP INTO SHIFT LOOP
LZA     CLA     REMR          RELOAD REMAINDER INTO AC
        TMI     )N
)P      TSX     BT,4          TEST FOR LEADING ZERO
        PZE     0,1           REMR (+), 2-BIT TEST
        TRA     LZB           = 1
        TRA     UPD           = 0, SHIFT OUT A LEADING ZERO
)N      TSX     BT,4          TEST FOR LEADING ZERO
        PZE     16384,0       REMR (-), 3-BIT TEST
        TRA     LZB           = 1
UPD     ALS     1             = 0, SHIFT OUT A LEADING ZERO
        STO     REMR          SAVE REMAINDER
        TMI     *+2           SENSE LIGHT 1 ON IF REMR IS (+)
        SLN     1             TURN ON THE SENSE LIGHT
        CAL     XSQ           MOVE LOW-ORDER SQUARE
        ARS     1
        SLW     XSQ           STORE NEW LOW-ORDER SQUARE
        CAL     DGLINE        INJECT CURRENT DIGIT
        SLT     1             TEST SENSE LIGHT 1
        ORS     TFR           REMR (-), 1 TO PARTIAL ROOT
```

```
        ARS    1              REMR (-). MOVE CURRENT DIGIT
        TIO    CHECK          TEST TO SEE IF FINISHED
        SLW    DGLINE         STORE NEW CURRENT DIGIT
        CAL    TFR            MODIFY TRIAL FACTOR
        ERA    DGLINE         ERASE OLD LOW-ORDER SQUARE
        ORA    XSQ            INJECT NEW LOW-ORDER SQUARE
        SLW    TFR            STORE NEW TRIAL FACTOR
        TXI    *+1,2,TS       UPDATE TIME UNIT COUNTER
        TRA    LZA            TRY AGAIN
LZB     TPL    NCORR          NO MORE LEADING ZEROS
        ADD    DGLINE         CORRECTION IF (-)
NCORR   STO    REMR
        TPL    NEXT           NEXT ITERATION IF (+)
NXT     TRA    NEXT           PERFORM NEXT ITERATION
NLZ     STO    REMR
        TRA    NEXT           PERFORM NEXT ITERATION
ERROR   CLA    IERROR         UPDATE ERROR COUNTER
        ADD    INT1
        STO    IERROR
        TRA    RSTR           STATUS QUO
NCHECK  CLA    ICHECK         UPDATE CHECK COUNTER
        ADD    INT1
        STO    ICHECK
        TRA    RSTR           STATUS QUO
        REM    CHECK THE RESULT
CHECK   CLA    TFR            MAKE ALLOWANCE FOR SHORT
        ANA    MSK            WIPE OUT EXCESS POSITIONS
        ALS    1              NORMALIZE FOR TEST PURPOSES
        STO    ROOT           FOR MULTIPLICATION
        XCA                   FOR MULTIPLICATION
```

```
        MPY     ROOT            SQUARE THE RESULT
        LRS     8               ROUND OFF
        RND
        ALS     8               REPOSITION
        STO     RTSQ            FOR COMPARISON
        SUB     OPR             OBTAIN DIFFERENCE
        ARS     10              SHIFT OUT MINIMUM ACCEPTABLE DIFFERENCE
        TNZ     NCHECK          DIFFERENCE TOO LARGE
        REM     FINISHED WITH THIS OPERAND
FINIS   NOP
RSTR    LXD     XRSV,1
        LXD     XRSV+2,4
        TOV     *+1
        TRA     2,4             RETURN
        REM     LEFT HALF BIT TEST ROUTINE
BT      STI     BTSV            SAVE INDICATORS
        STO     REMSV           SAVE REMAINDER
        PAI                     REMR TO INDICATORS
        CLA     1,4             SET UP INDICATOR TEST
        STT     *+2
        STA     *+1
        LNT     0               ON TEST FOR LEADING BIT
        SLN     2               = 0
        CLA     REMSV           = 1, REMR TO AC
        LDI     BTSV            RESTORE INDICATORS
        SLT     2               TEST FOR ON OR OFF
        TRA     2,4             = 1
        TRA     3,4             = 0
BTSV    BSS     1               SAVED INDICATORS
REMSV   BSS     1               SAVED REMAINDER
```

```
MSK     OCT     377777777600  MASK TO MAKE SHORT REGISTER
OPR     BSS     1             OPERAND FOR ANSWER CHECK
RFMR    BSS     1             REMAINDER REGISTER (AUGEND)
TFR     BSS     1             TRIAL FACTOR REGISTER (ADDEND)
DGLINE  BSS     1             DIGIT LINES
XSQ     BSS     1             LOW-ORDER SQUARE REGISTER
ROOT    BSS     1             NORMALIZED ROOT FOR ANSWER CHECK
RTSQ    BSS     1             SQUARED SQUARE ROOT
ENDBIT  OCT     000000000100  END MARKER
INT1    PZE     **1           FORTRAN INTEGER 1
XRSV    BSS     3             INDEX REGISTERS
*
RANDOM  COMMON  1             INITIAL RANDOM NO. FOR MPC
IERROR  COMMON  1             ERROR COUNTER
ICHECK  COMMON  1             CHECK FAILURE COUNTER
IQ      COMMON  500           TIMING DISTRIBUTION
NN      COMMON  1             NO. OF OPERANDS
XNN     COMMON  1             NO. OF OPERANDS
        END
```

Table A-9: Binary Square Root Simulation Program, Timing Distribution.

```
      SUBROUTINE INPUT2
      DIMENSION DUMMY(503)
      COMMON DUMMY
      COMMON NN,XNN
      COMMON K
20    FORMAT(I5)
21    FORMAT(I10,F10.0)
      READ INPUT TAPE 5,20,K
      READ INPUT TAPE 5,21,NN,XNN
      RETURN
      END
```

Table A-10: Input Routine, Timing Distribution.

```
      SUBROUTINE OUTPTZ
      DIMENSION JQ(500),Q(500)
      COMMON RANDOM
      COMMON IERROR,ICHECK
      COMMON JQ
      COMMON NN,XNN,K
3     FORMAT(54H1      STATISTICAL DISTRIBUTIONS FOR BINARY SQUARE ROOT)
33    FORMAT(22H0      NO. OF OPERANDS I6)
4     FORMAT(29H0      INITIAL RANDOM NUMBER  O12)
5     FORMAT(22H0      DENSITY FUNCTION)
6     FORMAT(I20,I5,F10.5)
7     FORMAT(38H0      CUMULATIVE DISTRIBUTION FUNCTION)
8     FORMAT(10F10.5)
9     FORMAT(29H0      NO. OF ERROR FAILURES  I4)
10    FORMAT(29H       NO. OF CHECK FAILURES  I4)
      XQ=0.0
      WRITE OUTPUT TAPE 6,3
      WRITE OUTPUT TAPE 6,33,NN
      WRITE OUTPUT TAPE 6,4,RANDOM
      WRITE OUTPUT TAPE 6,5
       DO 50 I=1,500
      IF(JQ(I))49,50,49
49    J=I+K
      XQ=JQ(I)
      XQ=XQ/XNN
      WRITE OUTPUT TAPE 6,6,J,JQ(I),XQ
50    CONTINUE
      DO 60 I=2,500
60    JQ(I)=JQ(I)+JQ(I-1)
```

```
      DO 61 I=1,500
      Q(I)=JQ(I)
61    Q(I)=Q(I)/XNN
      WRITE OUTPUT TAPE 6,7
      WRITE OUTPUT TAPE 6,8,(Q(I), I=1,500)
      WRITE OUTPUT TAPE 6,9,IERROR
      WRITE OUTPUT TAPE 6,10,ICHECK
      RETURN
      END
```

Table A-11: Output Routine, Timing Distribution.

DISTRIBUTION LIST

COPIES	AGENCY
2	Assistant Sec. of Def. for Res. and Eng. Information Office Library Branch Pentagon Building Washington 25, D.C.
10	Armed Services Technical Information Agency Arlington Hall Station Arlington 12, Virginia
2	Chief of Naval Research Department of the Navy Washington 25, D.C. Attn: Code 437, Information Systems Branch
1	Chief of Naval Operations OP-07T-12 Navy Department Washington 25, D.C.
6	Director, Naval Research Laboratory Technical Information Officer Code 2000 Washington 25, D.C.
10	Commanding Officer, Office of Naval Research Navy #100, Fleet Post Office New York, New York
1	Commanding Officer, O N R Branch Office 346 Broadway New York 13, New York
1	Commanding Officer, O N R Branch Office 495 Summer Street Boston 10, Massachusetts
1	Bureau of Ships Department of the Navy Washington 25, D.C. Attn: Code 607A NTDS
1	Bureau of Naval Weapons Department of the Navy Washington 25, D.C. Attn: RAAV Avionics Division
1	Bureau of Naval Weapons Department of the Navy Washington 25, D.C. Attn: RMWC Missile Weapons Control Div.
1	Bureau of Naval Weapons Department of the Navy Washington 25, D.C. Attn: RUDC ASW Detection & Control Div.
1	Bureau of Ships Department of the Navy Washington 25, D.C. Attn: Communications Branch, Code 686
1	Naval Ordnance Laboratory White Oaks Silver Spring 19, Maryland Attn: Technical Library

COPIES	AGENCY
1	David Taylor Model Basin Washington 7, D.C. Attn: Technical Library
1	Naval Electronics Laboratory San Diego 52, California Attn: Technical Library
1	University of Illinois Control Systems Laboratory Urbana, Illinois Attn: D. Alpert
1	University of Illinois Digital Computer Laboratory Urbana, Illinois Attn: Dr. J.E. Robertson
1	Air Force Cambridge Research Laboratories Laurence C. Hanscom Field Bedford, Massachusetts Attn: Research Library, CRX2-R
1	Technical Information Officer US Army Signal Research & Dev. Lab. Fort Monmouth, New Jersey Attn: Data Equipment Branch
1	National Security Agency Fort George C. Meade, Maryland Attn: R-4, Howard Campaigne
1	US Naval Weapons Laboratory Dahlgren, Virginia Attn: Head Compution Div., G.H. Gleissner
1	National Bureau of Standards Data Processing Systems Division Room 239, Bldg. 10 Attn: A.K. Smilow, Washington 25, D.C.
1	Aberdeen Proving Ground, BRL Aberdeen Proving Ground, Maryland Attn: J.H. Giese, Chief Compution Lab.
1	Commanding Officer O N R Branch Office John Crerar Library Bldg. 86 East Randolph Street Chicago 1, Illinois
1	Commanding Officer O N R Branch Office 1030 E. Green Street Pasadena, California
1	Commanding Officer O N R Branch Office 1000 Geary Street San Francisco 9, California
1	National Bureau of Standards Washington 25, D.C. Attn: Mr. R.D. Elbourn

COPIES	AGENCY
1	Naval Ordnance Laboratory Corona, California Attn: H.H. Weider
1	George Washington University Washington, D.C. Attn: Prof. N. Grisamore
1	Dynamic Analysis and Control Laboratory Massachusetts Institute of Technology Cambridge, Massachusetts Attn: D.W. Baumann
1	New York University Washington Square New York 3, New York Attn: Dr. H. Kallmann
1	Univ. of Michigan Ann Arbor, Michigan Attn: Dept. of Philosophy, Prof. A. W. Burks
1	Census Bureau Washington 25, D.C. Attn: Office of Asst. Director for Statistical Services, Mr. J.L. McPherson
1	University of Maryland Physics Department College Park, Maryland Attn: Professor R.E. Glover
1	Columbia University New York 27, New York Attn: Dept. of Physics, Prof. L. Brillouin
1	Hebrew University Jerusalem, Israel Attn: Prof. Y. Bar-Hillel
1	Massachusetts Institute of Technology Research Laboratory of Electronics Attn: Prof. W. McCulloch
1	University of Illinois Urbana, Illinois Attn: John R. Pasta
1	Naval Research Laboratory Washington 25, D.C. Attn: Security Systems Code 5266, Mr. G. Abraham
1	National Physical Laboratory Teddington, Middlesex England Attn: Dr. A.M. Uttley, Superintendent, Autonomics Division
1	Diamond Ordnance Fuze Laboratory Connecticut Ave. & Van Ness St. Washington 25, D.C. ORDTL-012, E.W. Channel
1	Harvard University Cambridge, Massachusetts Attn: School of Applied Science, Dean Harvey Brook

COPIES	AGENCY
1	Wright Air Development Division Electronic Technology Laboratory Wright-Patterson AFB, Ohio Attn: Lt. Col. L.M. Butsch, Jr. ASRUEB
1	Laboratory for Electronics, Inc. 1079 Commonwealth Ave. Boston 15, Massachusetts Attn: Dr. H. Fuller
1	Stanford Research Institute Computer Laboratory Menlo Park, California Attn: H.D. Crane
1	General Electric Co. Schnectady 5, N.Y. Attn: Library, L.M.E. Dept., Bldg. 28-501
1	The Rand Corp. 1700 Main St. Santa Monica, California Attn: Numerical Analysis Dept. Willis H. Ware
1	General Electric Research Laboratory P O. Box 1088 Schenectady, New York Attn: Information Studies Section R. L. Shuey, Manager
1	Stanford Research Institute Menlo Park, California Attn: Dr. Charles Rosen Applied Physics Laboratory
1	New York University New York, New York Attn: Dr. J.H. Mulligan, Jr. Chairman of E.E. Dept.
1	Marquardt Aircraft Company 16555 Saticoy Street P.O. Box 2013 - South Annex Van Nuys, California Attn: Dr. Basun Chang, Research Scientist
1	Texas Technological College Lubbock, Texas Attn: Paul G. Griffith Department of Electrical Engineering
1	L. G. Hanscom Field AF-CRL-CRRB Bedford, Mass. Attn: Dr. H.H. Zschirnt
1	Department of the Army Office of the Chief of Research & Development Pentagon, Room 3D442 Washington 25, D.C. Attn: Mr. L.H. Geiger
1	Bell Telephone Laboratories Murray Hill Laboratory Murray Hill, New Jersey Attn: Dr. Edward F. Moore
1	General Electric Research Lab. P.O. Box 1088 Schenectedy, New York Attn: V.L. Newhouse Applied Physics Section

COPIES	AGENCY
1	National Biomedical Research Foundation Inc. 8600 16th St., Suite 310 Silver Spring, Maryland Attn: Dr. R.S. Ledley
1	University of Pennsylvania Moore School of Electrical Engineering 200 South 33rd Street Philadelphia 4, Pennsylvania Attn: Miss Anna Louise Campion
1	Army Research Office OCR & D Department of Army Washington 2, D.C. Attn: Mr. Gregg McClurg
1	Mr. Paul W. Howerton Room 1053 M. Bldg. Code 163 CIA Washington, D.C.
1	University of Pennsylvania Mechanical Languages Projects Moore School of Electrical Engineering Philadelphia 4, Pennsylvania Attn: Dr. Saul Gorn, Director
1	Applied Physics Laboratory Johns Hopkins University 8621 Georgia Avenue Silver Spring, Maryland Attn: Document Library
1	Bureau of Supplies and Accounts, Chief Navy Department Washington, D.C. Attn: Code W3
1	Prof. E. L. Hahn Dept. of Physics University of California Berkeley 4, California
1	Auerbach Electronics Corporation 1634 Arch St. Philadelphia 3, Pa.
1	National Security Agency Fort George G. Meade, Maryland Attn: R. -42, R. Wiggington
1	Federal Aviation Agency Bureau of Research and Development Washington 25, D.C. Attn: RD-375 Mr. Harry Hayman
1	Federal Aviation Agency Bureau of Research & Development Center Atlantic City, New Jersey Attn: Simon Justman
1	Chief, Bureau of Ships Code 671A2 Washington, D.C. Attn: LCDR. E. B. Mahinske, USN
1	Lincoln Laboratory Massachusettes Institute of Technology Lexington 73, Massachusetts Attn: Library

COPIES	AGENCY
1	Dr. Tsute Yang Digital Systems Group Radio Corporation of America Moorestown, New Jersey
1	Professor C. L. Pekeris, Head Department of Applied Mathematics Weizmann Institute of Science Rehovoth, Israel
1	Mr. Julian H. Bigelow Institute for Advanced Study Princeton, New Jersey
1	Mr. Raoul Sajeva Uiale Legioni Romane 22/7 Milano, Italy
1	Electronics Research Laboratory University of California Berkeley 4, California Attn: Director
1	Mr. Gordon Stanley 7685 South Sheridan Ct. Littleton, Colorado Martin, Denver
1	R. Turyn Applied Research Lab. Sylvania El. Pd. Inc. 40 Sylvan Road Waltham 54, Mass.
1	P. Braffort CETIS Euratom C.C.R. Ispra (Varese), Italy
1	Information Processing Directorate Attn: C. J. Shaw System Development Corporation 2500 Colorado Avenue Santa Monica, California
1	Director Courant Inst. of Mathematical Sciences New York University 4 Washington Square New York 3, New York
1	Dr. Alston S. Householder Oak Ridge National Laboratory Oak Ridge, Tennessee
1	Dr. Milton E. Rose Lawrence Radiation Laboratory University of California Berkeley, California
1	Professor Maria G. Mayer University of California San Diego, California
1	Martin Graham Rice University Houston, Texas

COPIES	AGENCY
1	Institute for Defense Analysis Communications Research Division Von Neumann Hall Princeton, New Jersey
1	Major Wm. H. Harris Hdqrs. AFESD ESRDD L.G. Hanscom Field Bedford, Massachusetts
1	Dr. H. Goldstine, Director IBM, Thomas J. Watson Res. Ctr. P.O. Box 218 Yorktown Heights, New York
1	Fred Dion Rome Air Development Center Data Processing Branch Intelligence Processing Lab., RAWID Griffiss AFB, New York
1	Professor Ivan Flores South Huckleberry Drive Norwalk, Conn.
1	J. C. Murtha, Research Associate 490 West End Ave. New York 24, New York
1	Dr. J. Pomerene IBM Product Dev. Lab. Dept. 271 Poughkeepsie, New York
1	L. Freinkel U.S. Naval Ordnance Test Station Pasadena Annex 3202 E. Foothill Blvd. Pasadena 8, California
1	W. H. Rein C/O Institut für Praktische Mathematik Professor Dr. A. Walther Technische Hochschule Darmstadt, Germany
2	Atomic Energy Commission Research Division Germantown, Maryland Attn: Dr. C.V.L. Smith

Digital Technology